February 1990 Report No. STAN-CS-90-1304

A Model of Object-Identities and Values

by

Toshiyuki Matsushima and Gio Wiederhold

Department of Computer Science

Stanford University

Stanford, California 94305

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a REPORT SECURITY CLASSIFICATION	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Stanford University · ; 6b OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science Stanford, CA 94305	7b ADDRESS (City, State, and ZIP Code)
8a NAME OF FUNDING / SPONSORING ORGANIZATION DARPA ; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00039-84-C-0211
8c ADDRESS (City, State, and ZIP Code) Arlington, VA	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO ; PROJECT NO ; TASK NO ; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)

A Model of Object Identities and Values

12 PERSONAL AUTHOR(S)

Matsushima Toshiyuki

13a TYPE OF REPORT Research	13b TIME COVERED FROM 1988 TO 1990	14 DATE OF REPORT (Year, Month, Day) February 1990	15 PAGE COUNT

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

An algebraic formalization of the object-oriented data model is proposed. The formalism reveals that the semantics of the object-oriented model consists of two portions. One is expressed by an algebraic construct, which has essentially a value-oriented semantics. The other is expressed by object-identities, which characterize the essential difference of the object-oriented model and value-oriented models, such as the relational model and the logical database model. These two portions are integrated by a simple commutativity of modeling functions.
The formalism includes the expression of integrity constraints in its construct, which provides the natural integration of the logical database model and the object-oriented database model.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) ; 22c OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

A Model of Object-Identities and Values

Toshiyuki Matsushima, Gio Wiederhold

February 23, 1990

Abstract

In this report, a formalization of the object-oriented data model is proposed, which integrates value-oriented models and object-oriented models by providing a simple semantics of object-identity.

The formalism reveals that the semantics of the object-oriented model consists of two portions. One is expressed by an algebraic construct, which has essentially a value-oriented semantics. The other is expressed by object-identities, which characterize the essential difference of the object-oriented model from value-oriented models, such as the relational model and the logical database model. The value-oriented portion represents the abstraction of the real world objects, while the object-oriented portion represents the existence of the real world objects. These two portions are integrated by a simple commutative diagram of modeling functions.

The formalism includes the expression of integrity constraints in its construct of classes. which provides the natural integration of the logical database model and the object-oriented database model. More specifically, we will show that a datalog program can be expressed as a collection of classes in our model.

As an application of the formalism, formal guidelines on database design are also discussed.

Contents

1 Introduction

In recent years, many attempts have been made to formalize the semantics of the object-oriented model. As the result of these efforts, several models have been proposed [AK 89], [LR 89], [KW 89],[CW 89]. Roughly speaking, these models are logical database models with typed variables. Their approach is to incorporate a structured knowledge representation, such as complex objects, object-hierarchy, *into* a logical representation paradigm. However, the semantics of object-identity is not captured in these models. Although [AK 89] formalize object-identity in their model, the semantics remains complicated. Basically, what they have done is to "push" object-identity into a value-oriented framework consisting of logic and types. However, as discussed later, the notion of object-identity is something that will never fit into the value-oriented paradigm.

In this report, a formal semantics of an object-oriented model is proposed, which approaches the issue from the opposite direction. We try to incorporate a. logical knowledge representation *into* a structured knowledge representation paradigm. We will show that our approach provides a natural formalization of object-identity and a simple integration of the object-oriented paradigm and the value-oriented paradigm.

This report has two main objectives. One is to provide simple and elegant semantics of object-identity, which integrates value-oriented models and object-oriented models. The other is to extend the formalization of objects so that the integrity constraints are included.

1.1 Formalization of Object-identity

In this section, we first provide an overview of the origin and the role of object-identity in knowledge representation, using the discussions in the literature listed above. Then, we provide an outline of our formalization of object-identity.

The semantics of object-identity is obtained by considering a basic aspect of a knowledge representation. Namely, any knowledge representation is only an approximation of the real world knowledge. The existence of objects in the real world cannot be captured by the values of expressions. We consider an example. Let us assume that a concept 'person' is expressed by name and address according to the following *schema* in the sense of [AK 89][1].

$$Location\ = [city{:}String,\ street{:}String,\ number{:}Integer],$$

$$Person\ = [name{:}[first{:}String,\ last{:}String],\ address{:}Location].$$

In most cases, we can completely identify each individual person by providing the name and address. However, there is a possibility that two distinct persons with the same name are living at the same' place. The occurrence of these persons cannot be characterized by the values of attribut'es 'name' and 'address'. We can come up with two relevant solutions for this problem. One is to provide more attributes for expressing the concept 'person'. However, the *real* attributes of a person are almost infinite in number. So, even if we introduce many attributes for 'person', we cannot eliminate the possibility that some distinct persons are expressed by the same set of attribute values. The other solution is to provide a key attribute to express the uniqueness of each individual person. However, this does not provide a natural way of expressing the real world, because it is an artificial attribute. We cannot avoid the unnecessary semantics of the key attribute. For example, a. 'social-security-number' may be

[1] We use the notation explained in [AK 89] for the moment..

implemented as either an integer or a string consisting of digit characters. In order to define the equality of objects, we have to define it as equality of integer, or equality of string according to the "implementation." Further, we have to express the maintenance of the key attribute explicitly in the higher level semantics. For example, "Once an instance is created, the key attribute should not be altered", "there should not be more than one instance whose key attributes are identical." Since the semantics of "real existence of objects,' is just that of a. set with the equality relation, it is not desirable that the semantics of the implementation appears in higher level semantics.

The problem is essentially due to the inherent incompleteness of our representation. Therefore, rather than expressing the uniqueness of an occurrence in the real world by attribute values, we need something that specifies the existence of occurrence. The object-identity serves this role. It is important that an object-identity is not a value. Instead, it is an *entry point* for information access in our knowledge. In other words, it is the reference to knowledgebase. Hence, as discussed in [LR 89], it provides the basis for object sharing, which is the most important advantage of introducing object-identities in a practical system.

Let us come back to the previous example. Suppose that a person named "John Ford" lives at "2260 Yale Street Pa.10 Alto". Moreover, suppose that a person named "Mary Carter" lives with him. These facts are cspressed by:

$$`P001' = [name{:}[first{:}``John", last{:}``Ford"], address{:}`L010'],$$

$$`P002' = [name{:}[first{:}``Mary", last{:}``Carter"], address{:}`L010'],$$

$$`L010' = [city{:}``PaloAlto", street{:}``Yale", number{:}2260].$$

What happens if the name of *the* street where John lives is changed from "Yale" to "Harvard"? Since John lives at the location 'L010', the expression of the location becomes:

$$`L010' = [city{:}``PaloAlto", street{:}``Harvard", number{:}2260].$$

Hence, after the change, we can say that *both* John and Mary are living on Harvard Street. The point is that 'L010' corresponds to the existing location on earth, and John and Mary's address is expressed by *referring* to 'L010'. Thus, when its street name has been changed, the change is propagated properly.

So far, we have seen the origin of object-identity and the role of object-identity in the knowledge representation. To summarize:

- The object-identity corresponds to the real existence of objects in the real world, which cannot be captured by the the value of expression.
- The object-identity provides the basis of object-sharing. An object-identity is the reference to represented knowledge. which is exactly what is to be shared.

Next we claim that in order to take full advantage of object-sharing, attribute values of an object should be object-identities.

[AK 89], [CW 89] allow complex values[2] as the values of attributes. It provides us the complicated expression of objects. Namely, in the above example, $[first{:}``John", last{:}``Ford"]$ is a. complex(structured) value. However, this approach has a disadvantage. If we allow complex values, there is an inherent possibility that the subexpression of a complex value

[2] We use the term "complex value" instead of "complex object,". They don't carry object-identity.

would be changed. Since a substructure of a value cannot be shared. it will cause costly update maintenance. Of course, the schema is designed so that the attribute value of 'name' is really a value and not sharable, because it is quite natural to express a person's name as a value. However, even in this case, we can show an example that demonstrates the necessity of sharing objects.

Let us consider an additiona. concept, 'BusinessCard'.

$$BusinessCard = [company{:}String, title{:}String, name{:}[first{:}String, last{:}String]]$$

Assume that John's business card is expressed by:

$$`B011' = [company{:}``CDB", title{:}``salesman", name{:}[first{:}``John", last{:}``Ford"]]$$

What happens if John marries Mary and changes his last name to "Carter"? We have to create a new value:

$$[first{:}``John", last{:}``Carter"],$$

and replace

$$[first{:}``John", last{:}``Ford"].$$

The creation of the new value will be costly when the structure is large. Furthermore, we have to replace 'name' of both 'P001' and 'B011'.

If there is no need for the object-sharing, the comples value would be reasonable. However, if we have more than one concept that shares a same value, as in above example, we should incorporate with object-sharing. Thus, in this case, the following schema will be preferable.

$$Name = [first{:}String, last{:}String],$$

$$Person = [name{:}Name, address{:}Location],$$

$$BusinessCard = [company{:}String, title{:}String, name{:}Name].$$

The point is that *every attribute should refer to an object with object-identity.* Therefore, it is not desirable to design such a schema as the original 'Person' with complex-value [first: *String, last*:*String*] as attribute value. The schema must be changed dramatically when we add a new schema object like 'BusinessCard'.

In order to demonstrate the idea. more clearly, we repeat the discussion with the following schema. In this case, the attribute is not a comples value, but just a value.

$$Person, = [name{:}String, employer{:}String]$$

$$BusinessCard = [company{:}String, name{:}String].$$

The information about John will be expressed by:

$$`P001' = [name{:}``JohnFord", employer{:}``CDB"]$$

$$`B011' = [company{:}``CDB", title{:}``salesman", name{:}``JohnFord"]$$

If he changes his company from "CDB" to "HAL", we have to change the employer of 'P001' and the company of 'B011'. Therefore, rather than having value-attribute, we should have only attribute referring the object-identity of other objects. Namely,

$$`P001' = [name{:}`N012', employer{:}`E000'],$$

$$'B011' = [company:'E000', title:'S111', name:'N012'].$$

'N012' can be associated with a string value "CDB" or "HAL."

To summarize, in order to make use of object-sharing fully, it is preferable that a schema object doesn't have 'value' as an attribute value[3]. Instead, attribute value should be an object-identity referring to another object instance. In particular, it is not desirable to have complex values as attribute values.

Moreover, since the attribute names, such as 'name','employer', can be regarded as access functions, we get the following *flat* representation.

$$name('P001') = 'N012',\quad employer('P001') = 'E000',$$

$$company('B011') = 'E000',\ title('B011') = \ 'S111',\ name('B011') = \ 'P001'.$$

Thus the information about John is expressed by the partial functions from object-identities to object-identities. We call this represent ation space *object-identity space,* which will be precisely formalized in Section *4.2.* The semantics of this representation is quite simple.

However, the above representation does not have an important feature of object-oriented representation. That is *the explicit structural representation of knowledge.* One of the big advantages of frame or complex object in knowledge representation is that they provide the structure of knowledge that we can easily imagine and manage. Of course, we can espress the semantics of complex-object in first order logic by some transformation [CW 89]. However, if we express it in first order sentences or formulas, the structure is concealed in the semantics of sentences. Hence we have to interpret the first order sentences to get the structure. Therefore, we should integrate the object-identity space with a structured complex-value representation. In Chapter 4, we have a simple and elegant formalization that integrates them. The outline of the integration is as follows. First, we provide the syntactical construct of schema objects. Next, we provide the value-oriented model, i.e. an algebraic model with (complex) values. Then we provide the model expressed by the object-identity space. Finally, we provide the mapping that combines object-identity space and algebraic representation of complex-values. The compatibility of object-identity space representa tion and algebraic representation is expressed by a simple commutative diagram.

1.2 Integrity Constraints

In the conventional approach as [AK 89], [KW 89], schema objects are defined with the structure expressed by types. Then logical formulas are constructed on top of the objects (Rules in [AK 89], O-formulas in [KW 89]).

In our model, each schema object, called *C-class*, consists of *type* and a restriction *predicate.* The type expresses the structure of knowledge representation, which will be referred to as a comples object, a hierarchy of objects in conventional object-oriented models. The restriction predicate will espress the integrity constraint of the representation. Let us consider "absolute temperature,' as a. simple example. It can be expressed by the positive real numbers. The structure will be realized by the algebra R with operations +, −, ∗ etc. The integrity constraints will be expressed by the predicate $R(x) \equiv (x > 0)$ expressing "positiveness."

By including the integrity constraints as the basic component of each object, we can show that every unit. of knowledge can be expressed by objects. Even a logical formula can be

[3]The term "attribute value" is not a nice terminology. Maybe it should be "accessed attribute."

expressed by an object. In the conventional approach, a logical formula(ground fact) is a value in the sense that if every substructure of two logical formulas are the same, then those logical formulas are the same. However, as discussed in Chapter 5, even a logical formula cannot be treated as a value, due to the inherent incompleteness of our knowledge representation. Rather, it should be expressed by an *object that carries a unique object-identity.*

If we express knowledge by objects, we can provide a representation of the real world that is closer to our intuition than expressing know/edge by logical formulas on complex objects. We will discuss this matter in detail in Chapter 5.

1.3 Outline

The outline of this report is as follows.

In Chapter 2, we introduce a notion of *data algebra* that is an abstraction of data. Roughly speaking, the data algebra is the combination of type and integrity constraints. The type part is expressed by *a universal algebra,* and the integrity constraints part is espressed by a boolean function. The data algebra provides the basis for the semantics of value-oriented data. model, which is discussed in Chapter 4.

In Chapter 3, we introduce a notion of *C-class* that formalizes schema objects. A C-class is a construct that expresses a unit of real world knowledge. As mentioned earlier, in conventional models such as [AK 89], [KW 89], those units of knowledge are expressed by complex objects and logical formulas. The C-class is similar to class in the usual object-oriented languages, such as Smalltalk, and CLOS [WT 89]. A C-class is a combination of syntactical expressions of type and restriction predicate, the type specifies the structure and the restriction predicate expresses integrity constraints. A restriction predicate is a first order formula with implicitly typed variables, which is essentially a restricted form of O-formula [KW 89]. We also introduce a hierarchy among C-classes to espress the hierarchy of knowledge.

In Chapter 4, we discuss the main theme of this report, *object-identity.* First we formalize *a value-oriented model* of *C-classes.* Then we define a *object-oriented model* of *C-classes* by introducing the *object-identity space.* This object-oriented model represents the clear semantic distinction of a. value-oriented model and an object-oriented model. Further it clarifies the role of object-identity in the knowledge representation.

In Chapter 5, we consider the C-classes in detail and provide some kinds of C-classes. It reveals that even a logical representation of knowledge cannot be captured in a value-oriented paradigm. We discuss which knowledge should be value and which should be object as the database design issue. We introduce the *concept model* as a knowledgebase model.

In Chapter 6, we demonstrate the expressibility of the concept model, by simulating the semantics of other models, such as datalog, **IQL**[AK 89].

In Appendices, we briefly discuss database operations, inheritance and overloading. The semantics of database operation is quite simple, especially for queries. Furthermore, we provide the copy of the actual session performed on the prototype system that has been implemented.

2 Data Algebras

We introduce a notion of *data algebra* to express instances of schema objects. The notion of data algebra is an abstract formalization of complex objects with integrity constraints, which will serve as a *value-oriented model* of schema objects later. We assume a basic knowledge of the universal algebra, as found in p.22 - p.60 in [BS 81].

2.1 Multi-valued Universal Algebra

In order to define the notion of data algebra, we provide a precise definition of *multi-dued function*, *partial function*, and an extended universal algebra. If the reader does not like mathematical details, he/she may read only the last paragraph of this section.

Let A and B be sets, and let 2^A and 2^B be the power sets of A and B respectively. Then, a. function from 2^A to 2^B is called a *multi-dued function*[4] from A to B, if it satisfies the following condition.

$$\forall U \in 2^A,\ f(U) = \bigcup_{x \in U} f(\{x\}).$$

We denote the multi-valued function as:

$$f{::}A \rightharpoonup B.$$

It is easily proven that the composition of multi-valued functions is also a multi-valued function. Namely,

$$f{::}A \rightharpoonup B, g{::}B \rightharpoonup C \Rightarrow g \circ f{::}A \rightharpoonup C.$$

A multi-valued function f is called *total* if

$$f{::}A \rightharpoonup B, \quad \forall x \in A,\ f(\{x\}) \neq \emptyset.$$

Note that we can construct a category consisting of sets as objects and multi-valued functions as morphisms. The identity multi-valued function id_A on a set A is the identity function on 2^A.

For a multi-valued function f from A to B, *we* can always define the *quasi-inverse function* f^{-1} from B to A.

$$\forall V \in 2^B,\ f^{-1}(V) \stackrel{def}{=} \{x \in A \mid (f(x) \cap V) \neq \emptyset\}.$$

A multi-valued function f from A to B is called *injective* if f^{-1} of equals id_A. The function f is called *surjective*, if $f \circ f^{-1}$ equals id_B.

A partial function f from A to B is a multi-valued function from A to B such that for each clement of A, the cardinality of its image is no more than one,

$$\forall x \in A, \quad card(f(\{x\})) \leq 1.$$

The *domain* $\partial(f)$ of a partial function f is:

$$\partial(f) = \{x \in A \mid f(\{x\}) \neq \emptyset\}.$$

Any *function* h from A to B can be regarded as a multi-valued function. Namely, we can define a multi-valued function $\hat{h}$ by:

$$\forall U \in 2^A,\ \hat{h}(U) = \{f(x) \mid x \in U\}^5.$$

[4]A multi-valued function from A to B is equivalent to a binary relation on $A \times B$.

[5]The operator . can be considered as a functor from the category of sets to another category consisting of sets as objects and multi-valued functions as morphisms

In the rest of this report, we use the following simplified notation so long as it causes no confusion. For a multi-valued function from A to B, for an element x of A, and y of B,

$$f(x) \overset{def}{\equiv} f(\{x\}), \quad (f(x) = y) \overset{def}{\equiv} (f(\{x\}) = \{y\}).$$

Moreover, we introduce a virtual element $\nu_\perp$ to express "undefinedness", which is called the *null value*. The null value $\nu_\perp$ is a. common element of all sets. For a multi-valued(pa.rtia.1) function f, we denote

$$f(x) = \nu_\perp,$$

if

$$f(\{x\}) = \emptyset.$$

Now we extend the notion of universal algebra. A multi-valued universal algebra A is a pair of a set A and a family $\{f_i\}_{i\in I}$ of multi-valued functions. All the notions, such as homomorphism, isomorphism, are redefined using multi-valued functions instead of functions. Similarly, a. *partial-valued universal algebra* is a multi-valued universal algebra such that all the functions are partial.

The notion of data algebra. is defined by multi-valued universal algebras. However, in order to make the discussion simple, we only consider partial-valued universal algebras in the rest of this report. The reader can consider the partial-valued universal algebra as usual universal algebra, except for the existence of null value. Hence, we use the term "universal algebra" instead of "partial-valued universal algebra" from now on. But readers should remember that functions are partial.

2.2 Definition of Data Algebra

In this section, we provide the definition of the data algebras. A data algebra δ is a pair of a universal algebra [6] A and a *restriction function* [7]. Namely,

$$\delta = (\mathbf{A}, \mathbf{r}), \ \mathrm{A} = (\mathrm{A}, \{f_i\}_{i\in I}), \ \mathrm{r} : \mathrm{A} \to 2,$$

where 2 is the two-element boolean algebra.,

$$\mathbf{2} = (\{0, 1\}, \wedge, \vee, \neg).$$

Further, we assume that each data algebra contains a. special element null *value* $\nu_\perp$. As mentioned before, the null value expresses "undefinedness." For each function, if one of the arguments is null value then its value is also null value.

A data algebra is the abstraction of a collection of data with operations on it. For example, "positive numbers" would be expressed by a data algebra:

$$(\mathbf{R}, \mathrm{r}), \ \mathrm{r}(\mathrm{x}) \overset{def}{=} \begin{cases} 1 & (x > 0) \\ 0 & (x \leq 0), \end{cases}$$

where R is the universal algebra of real numbers.

[6]We should remember that this universal algebra is a partial-valued universal algebra defined in the previous section. We can regard it, as if it was a usual universal algebra by introducing a null-value $\nu_\perp$ as a common value ot all universal algebra.

[7]More precisely, **r** is a function on the *domain* of A. However, we describe it as a function on A. Similarly, throughout this report, we treat A and its domain interchangeably so long as the meaning is clear. For example, for give universal algebras A, B, we would state something like "a mapping from A to B". The meaning is " a mapping from a domain of A to the domain of B."

2.3 Fundamental Operators

We introduce operations among data algebras. These operations will provide the interpretations of fundamental operations on C-classes. which will be introduced in Chapter 3. Each operation happens to have a corresponding construct in relational algebra or SQL. However, we should note that these operators have not been obtained by a mere extension of relational algebra, but by the consideration of knowledge representation, as their names suggest. We could say that one of the reasons of the success of relational model is due to the fact that the relational operations have a. correspondence to a higher level of mental processes, such as abstraction of concepts. This will be clear when we introduce the fundamental operators on C-classes in Chapter 3.

2.3.1 Aggregation

The aggregation operator constructs a complex structure out of data algebras. It. is similar to Cartesian product operator in relational algebra.

Let Φ be a set of symbols, and let α be a. mapping from Φ to a set of data algebras,

$$\alpha(f) = \delta_f = (\mathbf{A}_f, \mathbf{r}_f) \quad (f \in \Phi).$$

Then the *aggregation* of $\{\delta_f\}_{f\in\Phi}$ is

$$(\prod_{f\in\Phi} \mathbf{A}_f, \bigwedge_{f\in\Phi} (\mathbf{r}_i \circ \mathbf{p}_i))$$

where $\mathbf{p}_i$ is the projection from $\prod_{i=1}^{n} \mathbf{A}_i$ to $\mathbf{A}_i$, and $\circ$ designates a composition of mappings. We denote the aggregation by

$$\prod(\Phi, \alpha) \text{ or } \prod_{f\in\Phi} \delta_f.$$

In particular, if Φ is equal to $\{1, . . . , n\}$, we denote it

$$\prod_{i=1}^{n} \delta_i .$$

Moreover? if δ_i is equal to a data algebra S for each i $(1 \leq i \leq n)$, we denote δ^n instead of $\prod_{i=1}^{n} \delta_i$. Furthermore, if we write the aggregated data algebras as:

$$\delta'' \times \delta, \delta'' \times \delta \times \delta',$$

for given data algebras S, δ', δ'', etc, it means that we are assuming the following implicit sequencing,

$$\alpha(1) = \delta'', \alpha(2) = \delta, \alpha(3) = \text{s'}. . . . .$$

2.3.2 Recursive Aggregation

In order to provide an algebraic model for recursive types, we introduce *recursive aggregation.* Let G be a directed graph with nodes I;' and labeled edges E,

$$\text{G} = (V, \text{E}).$$

We denote an element of E by (n, m, l), which means that there is an edge from n to m labeled by l. Further, let α be a mapping from U to a set of data algebras, where U is the subset of V such that each element u of U doesn't have any edge that comes into u,

$$U \overset{def}{=} \{u \in V \mid \neg(\exists v \exists l\ (v, u, l) \in E)\},$$

$$\alpha(n) = (\mathbf{B}_n, \mathbf{s}_n)\ (n \in U).$$

Then, the recursive aggregation with respect to G and α is defined as follows.

$$\prod(G, \alpha) \equiv (\prod_{n \in V} \mathbf{A}_n,\ \bigwedge_{n \in V}(\mathbf{r}_n \circ \pi_n))$$

$$\forall n \in V,\ \mathbf{A}_n \overset{def}{=} \begin{cases} \mathbf{B}_n & (n \in U) \\ \prod_{(n,m,l)\in E} \mathbf{A}_{(m,l)} & (n \notin U) \end{cases}$$

$$\forall (n, m, l) \in E\ \mathbf{A}_{m,l} = \mathrm{A},$$

$$\forall n \in V,\ \mathbf{r}_n(x) \overset{def}{=} \begin{cases} 1\ (1 \in \mathbf{t}_n(x)) \\ 0\ (\text{otherwise}) \end{cases}$$

The functions $\mathbf{t}_n$ are multi-valued functions from A_n to $\mathbf{2}$, which is defined as follows.

$$\mathbf{t}_n \overset{def}{=} \begin{cases} \bigwedge_{(n,m,l)\in E}(\mathbf{s}_m \circ \pi_{(m,l)}) & (n \notin U) \\ \mathbf{s}_n & (\text{otherwise}) \end{cases}$$

$$\text{where } \mathbf{s}_m(x) = \begin{cases} 1 & (x = \nu_\perp) \\ \mathbf{t}_n(x) & (\text{otherwise}) \end{cases}$$

Note that the elements of $\mathbf{A}_n (n \notin U)$ have an infinite structure in general. We may regard those elements as infinite trees. However, since we allow null value $\nu_\perp$ as the common element of every algebra, we can espress elements with a. finitely recursive structure. The function $\mathbf{t}_n$ is well-defined, if the recursive definition assigns consistent values to each subtrees. Although the restriction function r is a partial function, it is well-defined on the elements with finite structure and cyclic structure. The aggregation defined above is a special case of the recursive aggregation. In fact, if we assume:

$$V = \Phi,\ E = \emptyset, \alpha(f) = \mathbf{A}_f (f \in \Phi),$$

we get the original aggregation operator.

2.3.3 Abstraction

The abstraction operator constructs a new data algebra ignoring some of the substructures of a data algebra. It is similar to the projection operator in relational algebra.

For $f \in \Phi$, let $\mathbf{A}_f$ be a universal algebra., where Φ is a set of symbols, and let Ψ be a subset of Φ. Let us consider the following data algebra δ

$$\delta = (\prod_{f \in \Phi} \mathbf{A}_f,\ \mathbf{r}).$$

where $\prod_{f\in\Phi} \mathbf{A}_f$ is the product algebra of $\{\mathbf{A}_f\}_{f\in\Phi}$. Further, let P_Ψ be the projection from $\prod_{f\in\Phi} \mathbf{A}_f$ to $\prod_{g\in\Psi} \mathbf{A}_g$. The abstraction $\Upsilon(\delta, \Psi)$ of the data algebra S with respect to Ψ is defined as:

$$\Upsilon(\delta,\Psi) = (\prod_{g\in\Psi} \mathbf{A}_g, \tilde{\mathbf{r}}),$$

where

$$\tilde{\mathbf{r}}(x) = \begin{cases} 1 \ (if\ \exists y \in P_\Psi^{-1}(x), \mathbf{r}(y) = 1) \\ 0 \ (\text{otherwise}) \end{cases}$$

2.3.4 Restriction

The restriction operator imposes a new restriction on a data algebra. It is similar to the selection operator in relational algebra.

Let S = $(\mathbf{A}, \mathbf{r})$ be a data algebra, and let s be a mapping from the domain of A to 2. Then the restriction with respect to s is

$$(\mathbf{A}, \mathbf{r} \wedge \mathrm{s})$$

The restriction is denoted by $\Theta(\delta, \mathrm{s})$.

2.3.5 Sequence Construction

The sequence construction operator constructs a data algebra consisting of sequences of elements of a data algebra.

Let S = $(\mathbf{A}, \mathbf{r})$ be a data algebra. The sequence algebra $\mathbf{Seq}(\delta)$ derived from S consists of the direct sum[8] of the product algebra A'" $(i = 0, 1, 2, \ldots)$, and the relevant restriction function $\mathbf{r}_{seq}$,

$$\mathbf{Seq}(\delta) = (\Sigma_{n=0}^{\infty}\mathbf{A}^n, \ \mathbf{r}_{seq})$$
$$\forall x = (x_1, x_2, \ldots, x_n) \in \mathbf{Seq}(\delta), (n = 0, 1, 2, \ldots)$$
$$\mathbf{r}_{seq}(x) = \begin{cases} \bigwedge_{i=1}^{n} \mathbf{r}(x_i) & (n > 0) \\ 1 & (n = 0) \end{cases}$$

Given a class of universal algebras. The set of finite sequences of elements of the algebras in the class forms a universal algebra. with functions, *length, concutennte, null*, reverse, etc. We designate it by SEQ. We assume that the direct sum $\Sigma_{n=0}^{\infty}\mathbf{A}^n$ is a subalgebra of SEQ by embedding it in **SEQ**.

2.3.6 Bag Construction

The bag construction operator constructs a data algebra consisting of bags of elements of a data algebra..

Let S = (A? **r**) be a data, algebra. We can define a congruence relation $\sim$ in the direct sum algebra. $\Sigma_{n=1}^{\infty}$ A'" as follows. For elements $\vec{x}$, $\vec{y}$ of $\mathbf{Seq}(\delta)$,

$$\vec{x} = (x_1, \ldots, x_n), \ \vec{y} = (y_1, \ldots, y_m),$$

[8]The direct sum is always a partial-valued algebra.

the sequences $\vec{x}$ and $\vec{y}$ are equivalent with respect to $\sim$; $\vec{x} \sim \vec{y}$, if n equals m, and there exists a permutation σ of order n such that

$$(x_1, \ldots, x_n) = (x_\sigma(1), \ldots, x_{\sigma(n)}).$$

Then the *bag algebra* derived from δ consists of the quotient algebra of $\Sigma_{n=0}^{\infty}\mathbf{A}^n$ with respect to $\sim$ and the restriction function $\mathbf{r}_{bag}$. Since the restriction function $\mathbf{r}_{seq}$ of $\mathbf{Seq}(\delta)$ has the same value on the equivalence class of $\sim$, we can define the restriction function $\mathbf{r}_{bag}$ of $\mathbf{Bag}(\ \delta)$ by:

$$\mathbf{r}_{bag}([\vec{x}]) = \mathbf{r}_{seq}(\vec{x}),$$

where $[\vec{x}]$ is the equivalence class with respect to $\sim$ containing $\vec{x}$. Similarly, we can construct a universal algebra BAG as a quotient algebra of SEQ. As in the definition of $\mathbf{Seq}(\delta)$, we assume that the algebraic part of $\mathbf{Bag}(\delta)$ is a subalgebra of BAG by embedding it in BAG.

2.3.7 Set Construction

The set construction operator constructs the data algebra consisting of finite sets of elements of a data algebra..

Let δ be (A, r). The set algebra Set(d) is the collection of finite elements of A that satisfies r. The definition is as follows. First, we define a restriction function s on Bag(G). We denote an element of Bag(6) by $[\vec{x}]$, where $\vec{x}$ is an element in $\mathbf{Seq}(\delta)$. Then.

$$\vec{x} = (x_1, x_2, \ldots, x_n)$$
$$\mathrm{s}([\vec{x}]) \overset{def}{=} \begin{cases} 1 & (\text{if } (i \neq j \Rightarrow x_i \neq x_j)\ ) \\ 0 & (\text{otherwise}) \end{cases}$$

Nest. let SET be the universal algebra of finite sets with functions $\cup$(union), $\cap$(intersection), -(difference), etc. Then set algebra of *Set(G)* is obtained from $\Theta(\mathbf{Bag}(\delta), \mathrm{s})$ by regarding its algebraic component as subalgebra of SET. Namely,

$$\Theta(\mathbf{Bag}(\delta), \mathrm{s}) = (\Sigma_{n=0}^{\infty}\mathbf{A}^n/\sim, \mathbf{r}_{bag} \wedge \mathrm{s}).$$

2.3.8 Categorization

The categorization operator constructs a new data algebra by categorizing elements of a data algebra. with respect to the values of some substructures. It is similar to the grouping construct of SQL without aggregation functions.

Let Φ be a set of symbols, let Ψ be a subset of Φ and. let Ψ^c be the complement of Ψ,

$$\Psi \subset \Phi, \quad \Psi^c = \Phi - \Psi.$$

Further let β be a mapping from Φ to a set of universal algebra. Now let us consider the following data. algebra δ.

$$\delta = (\prod_{f \in \Phi} \beta(f), \mathbf{r}).$$

Then the *categorization* $\Omega(\delta, \Psi)$ of δ with respect to Ψ is:

$$\Omega(\delta, \Psi) = \Theta(\Upsilon(\delta, \Psi) \times \mathbf{Set}(\Upsilon(\delta, \Psi^c)), \mathbf{r}_\Omega).$$

The restriction function $\mathbf{r}_\Omega$ is defined as follows.

$$\mathbf{r}_\Omega((x, y)) = \begin{cases} 1 & (\forall z \in y, \mathbf{r}(x \oplus z) = 1) \\ 0 & \text{(otherwise)}, \end{cases}$$

where for $(x_1, \ldots, x_n)$ in $\prod_{f \in \Psi} \beta(f)$, $(y_1, \ldots, y_m)$ in $\prod_{g \in \Psi^c} \beta(g)$,

$$(x_1, \ldots, x_n) \oplus (y_1, \ldots, y_m) = (x_1, \ldots, x_n, y_1, \ldots, y_m) \in \prod_{f \in \Phi} \beta(f).$$

2.4 Many-Sorted Data Algebra

So far, we have introduced the notion of data algebra based on universal algebras. In this section, we extend the notion to many-sorted universal algebras instead of universal algebras.

First we consider a many-sorted algebra with sorts S. For a sort s in S, let us denote the universal algebra of the sort s by A,, and let $\mathcal{A}(s)$ be the collection of all subalgebra of A,. Further let d(S) be the closure of $\cup_{s \in S}$ d(s) with respect to the Cartesian product operator.

A set of data algebras D is the *many-sorted data algebras* with sorts S, if

$$\forall \delta \in \mathbf{D}\ \delta = (\mathbf{A}, \mathbf{r}),\ \mathrm{A} \in \mathcal{A}(S).$$

We call the data algebra of the following form as the *primitive data algebra* of sort s.

$$\delta = (\mathrm{A},, \mathbf{r})\ (s \in S).$$

We assume that any primitive data algebra of sort s will never be derived from primitive algebras of different sorts with fundamental operators. Namely, any data algebra of the form (A,, $\mathbf{r}$) (s $\in$ S) will never be derived from another data algebras of the form (A',, $\mathbf{r}$) ($s' \in S - \{s\}$) with fundamental operators.

From now on, we assume that data algebras axe constructed on a many-sorted algebra, even if the sorts S is not specifically stated. In another word, data algebras are *generated* from primitive data algebra in the sense defined in the nest section.

2.5 Generated Data Algebra

For a given set of data algebras, we can generate data algebras by the fundamental operators, such as aggregation, restriction, abstraction etc. We call a set of data algebras *algebraic family of data algebras* if it is closed under these operations. For a set D of data algebras, we can consider the minimum algebraic family of data algebras that contains D. We call it the *algebraic closure* of D and denote it as $\overline{\mathbf{D}}$. Since the intersection of algebraic families is also an algebraic family, it is obvious that there exists a unique algebraic closure for any set of data algebras. In fact, the closure of D is the in tersection of all algebraic families that contain D.

Conversely, we can consider the minima.l set of data algebras that generate a given set D of data algebras. More precisely, we can consider the set $\kappa(\mathbf{D})$ of data algebras such that:

- the algebraic closure of $\kappa(\mathbf{D})$ contains D.

$$\overline{\kappa(\mathbf{D})} \supset \mathrm{D}.$$

- among the sets that satisfy the above condition, h;(D) is minimal. Namely, for a set of data algebras E, if

$$\overline{\mathrm{E}} \supset \mathrm{D} \text{ and } \kappa(\mathbf{D}) \supset \mathrm{E},$$

then

$$\mathrm{D} = \mathrm{E}.$$

It is not difficult to prove the uniqueness of $\kappa(\mathbf{D})$ up to isomorphism, if D is finite. Namely, it is not only minimal but also minimum. So we call it the *kernel* of D. The kernel of a set of data algebras will provide the building bricks to construct the data algebras.

2.6 Named Data Algebra

The notion of data algebra will provide a structure of the space to express our knowledge. However, the structure itself is not enough. For example, we can express "absolute temperature" and "half line" by the same data. algebra as "positive numbers", which is defined in section 2.2 as an example. Moreover, we don't want to allow operations such as:

$$1^\circ\mathrm{K} + 2cm.$$

Thus we need to distinguish the data, algebras that are espressions of "absolute temperature" and "half line." Hence, we attach names to all algebras to distinguish them. We don't allow algebraic operations between the elements of data algebras with different names. A *named algebra* is expressed by a tuple:

$$(n_\delta, \mathbf{A}_\delta, \mathbf{r}_\delta).$$

In the rest of this report, we assume that every data algebra is named. However, when we don't have to consider the name explicitly, we use the previous notation without a name.

2.7 Hierarchy of Data Algebras

In the later chapters, we will see that data algebras play the role of model of a knowledge representation (schema representation). In order to express the hierarchy of knowledge. we introduce mappings among data algebras. First we assume that there esists a partial order $\preceq$ among names of data algebras. If $n \preceq n'$, we say that. the name n is a. *subname* of n'. A *subtype mapping* is the mapping from a data algebra to another data algebra, which is defined as follows.

Let us consider the many-sorted data algebra on the sort S. Let d be the set of universal algebras corresponding to the sorts.

$$\mathrm{d} = \{\mathrm{A}_s \mid s \in S\}.$$

Let. D be the many-sorted data algebra on S.

$$\mathrm{D} = \{\delta = (n_\delta, \mathbf{A}_\delta, \mathbf{r}_\delta)\}.$$

A subtype mapping ρ from a data algebra δ to another data algebra δ' is the mapping that satisfies the following conditions. Let us assume that:

$$\delta = (n, \mathbf{A}, \mathbf{r}),\ \delta' = (n', \mathbf{A}', \mathbf{r}').$$

- Case 1: The data algebras δ and δ' are primitive algebras.
  - The name n is a subname of n' and the algebras are the same.

$$n \preceq n' \textit{ and } A = A',$$

  - The restriction function r is stricter than r' and the ρ is the inclusion mapping. Namely,

$$\forall x \in \mathbf{A}, \mathbf{r}(x) = 1 \Rightarrow \mathbf{r}'(x) = 1,$$

$$\partial(\rho) = \{x \in \mathbf{A} \mid \mathbf{r}(x) = 1\},$$

$$\forall x \in \partial(\rho),\ \rho(x) = x.$$

- Case 2: The data algebras δ and δ' are compound algebras:

$$\delta = (n, \prod_{i \in \Phi} \mathbf{A}_i, \mathbf{r}),\ \delta' = (n', \prod_{i \in \Phi'} \mathbf{A}'_i, \mathbf{r}').$$

  - The name n is a subname of n'; $n \preceq n'$.
  - The attribute Φ' is a subset of Φ; $\Phi' \subseteq \Phi$.
  - For each f in Φ', there exists a subtype mapping ρ_f from $(\mathbf{A}_f, \widetilde{\pi_f}(\mathbf{r}))$ to $(\mathbf{A}'_f, \widetilde{\pi_f}'(\mathbf{r}'))$, where
    $\widetilde{\pi_f}(\mathbf{r})(x) = \begin{cases} 1 & (\text{if } \exists y\ (\pi_f(y) = x) \wedge (r(y) = 1)) \\ 0 & (\text{otherwise}), \end{cases}$
    π_f is the projection from $\Pi_{g \in \Phi} \mathbf{A}_g$ to $\mathbf{A}_f$.
    Similarly for $\widetilde{\pi_f'}(\mathbf{r}')$.

  - Let Π be the projection from $\Pi_{f \in \Phi}\ \mathbf{A}_f$ to $\Pi_{f \in \Phi'}\ \mathbf{A}_f$. Then,

$$\forall x \in \Pi_{f \in \Phi}\ \mathbf{A}_f,\ \mathbf{r}(x) = 1 \Rightarrow \mathbf{r}'((\pi_{f \in \Phi'} \rho_f) \circ \Pi(x)) = 1,$$

  where $\pi_{f \in \Phi'} \rho_f$ is the product mapping:

$$\forall x \in \Pi_{f \in \Phi'}\ \forall g \in \Phi',\ \pi_g((\pi_{f \in \Phi'} \rho_f)(x)) = \rho_f(\pi_g(x)).$$

  - The subtype mapping ρ from δ to δ' is defined by:

$$\rho = (\pi_{f \in \Phi'} \rho_f) \circ \Pi.$$

We say that δ is a *subtype algebra* of δ' if there exists a subtype mapping from δ to δ'.
If there exists a subtype mapping from δ to S', δ' will be a model of a more general concept than the concept that has the model δ. We will discuss it precisely in Chapter 3.

3 C-Classes

In order to formalize the construct of schema objects, we introduce the notion of *C-class*[9]. First we define the structure of C-classes.

[9]C-class is a kind of class. The letter C in "C-class" is intended to suggest *concept*.

3.1 C-Class Construct

The set Γ of *C-classes* is defined as follows.

3.1.1 Definition of C-Classes

$$\Gamma = \{\gamma \mid \gamma = (n_\gamma,\ \Phi_\gamma,\ v_\gamma,\ T_\gamma,\ \Delta_\gamma,\ R_\gamma)\ \}.$$

The intended meaning of symbols is:

- The name n_γ of γ is a symbol that designates the name of the C-class γ. The symbol is unique to each C-class.
- The attributes set Φ_γ of γ is a set of function symbols that designate attribute names.
- The attribute value v_γ of γ is a mapping from Φ_γ to the set of C-class names in I'.
- The structural sentences T_γ of γ are a set of sentences that define the algebraic structure of a universal algebra, which specifies the structure of the representation.
- The auxiliary sentences A, of γ is a set of sentences that defines new functions and predicates concerning γ. A, is used to simplify the espression.
- The restriction formula R_γ of γ is a well-formed formula with one free variable. This formula specifies a subset of the domain of the universal algebra defined by T_γ. It is the restriction condition on the domain.

The above construction provides a language for conceptualization of the real world. But we should keep in mind that our conceptualization is always incomplete. Since any object in the real world has almost infinitely many attributes, our conceptualization of the object will be only an approximation. We should distinguish between "real conceptual world" and "our conceptualization." The real conceptual world is the complete conceptualization of the real physical world. In the real conceptual world, a concept can be characterized by the set of attributes. Namely, any two distinct concepts have different sets of attributes. However our conceptualization may not be complete, two distinct concepts may be expressed with identical attributes. Therefore we need C-class names to identify each distinct concept. (It is true that we can carefully choose attribute names Φ_γ so that any distinct concepts are expressed with different attributes in our conceptualization. However, it becomes fairly difficult to design schema in such a way, if the schema is big. Moreover, if the schema will change in the course of time, the maintenance of consistent attribute names will be much more difficult.)

3.1.2 Examples of C-classes

We use the prefix notation for +,-, > etc., instead of the conventional infix notation. The only exception is equality =.

- Integer In our model, we treat integers as the instances of a C-class.

  $Integer = (integer,\ \emptyset,\ \perp,\ T_{Integer},\ \Delta_{Integer},\ TR\ UE)$,
  where
  $T_{Integer} = \{\forall x\ \forall y\ t\ (x, y) = +(y, x),$
  $\qquad \forall x\ \forall y\ \forall z\ t\ (x, +(y, z)) = +(+(x, y), z),$
  $\qquad$ etc.$\}$,

$A_{Integer} = \{\forall x\ Positive(x) \equiv > (x, 0), etc.\}.$

- People

  People is conceptualized by name and age in this example.

$$Person = (person, (name, age), v_{person}, T_{person}, \Delta_{person}, R_{person}).$$

$$v_{person}(name) = string,\ v_{person}(age) = integer,$$

  where *string* and *integer* designate the C-classes that have algebraic structure of strings and integers,

$$\begin{aligned} T_{person} = \{ & \forall x T(name(x), string) \wedge T(age(x), integer), \\ & \forall x \forall y\ name(modify(x, person, y)) = y, \\ & \forall x \forall y\ age(modify(x, person, y)) = y\}. \end{aligned}$$

  where *modify* designates the function that modifies the attribute values of C-classes.

$$\Delta_{person} = \{\forall x\ OldPerson(x) \Leftrightarrow age(x) > 60, etc\},$$

$$R_{person}(x) \equiv ((0 \le age(x) \le 200) \wedge \ldots).$$

- Rational Numbers

  The structured values, such as rational numbers, are also expressed by instances of a C-class. The expressions of rational numbers are expressed by:

  $Rationales = (rational, \{num, den\}, v_{Rational}, T_{Rational}, \Delta_{Rational}, R_{rational}),$
  where
  $v_{Rational}(num) = v_{Rational}(den) = integer,$
  $T_{Rational}$

$$\begin{aligned} = \{ & \forall a \forall b\ num(a) = x \wedge num(b) = u \wedge den(n) = y \wedge den(b) = v \\ & \Rightarrow \\ & num(+(a, b)) = +(*(x, v), *(u, y)) \wedge den(+(a, b)) = *(y, v), \\ & etc.\}, \end{aligned}$$

  $\Delta_{Rational} = \{\forall x\ Invertible(x) \equiv \neg(num(x) = 0), etc\}.$
  $R_{rational}(x) = \neg(den(x) = 0).$

- Set -of-Integer

  A set of a concept is expressed as a C-class without attributes. We assume that a. predicate symbol `T` is provided to designate the instance-class relation. We also assume that each set C-class has a standard predicate In, such that `In(x,y)` means x is in a. set y. We will extend this example to a general case later.

  $Set\text{-}of\_Integer = (set\text{-}of\text{-}integer.\ \emptyset, \bot, T_{Set}, \Delta_{Set\_of\_Integer}, R_{Set\_of\_Integer}),$
  $T_{Set} = \{\cup(x, y) = \cup(y, x), \cap(x, \cup(y, z)) = \cup(\cap(x, y), \cap(x, z)), etc.\},$
  $R_{Set\_of\_Integer}(x) = (\forall y\ \mathrm{In}(y, x) \Rightarrow \mathrm{T}(y, integer)).$

$$\begin{aligned} \Delta_{Set\_of\_Integer} = \{ & \forall x\ One\text{-}Element(x) \\ & \Leftrightarrow \\ & (\forall y \forall z\ \mathrm{In}(y, x) \wedge \mathrm{In}(z, x) \Rightarrow y = z), etc.\}, \end{aligned}$$

In above examples, we have introduced relation symbols T and In. From now on. we assume these symbols are part of the basic construct of C-classes.

3.1.3 Primitive C-Classes

Let us consider a concept with some attributes. We may say the concept is constructed by the concepts that are attribute values of the concept. To formalize this intuition, we impose a condition on the structural sentence T_γ of the C-class with non-empty attributes. If it is not specially declared, we assume that any C-class with non-empty attributes has the structural sentences T_γ containing the following sentences T^0_γ,

$$T^0_\gamma \subseteq T_\gamma.$$

Let Φ_γ be $\{ f_1, \ldots, f_n\}$, then

$$\begin{aligned} T^0_\gamma \;=\; & \{ \forall x \forall y\, f_i(\mathit{modify}(x, f_i, y)) = y \mid i = 1, \ldots, n \} \cup \\ & \{ \forall x\, T(f_i(x), v_\gamma(f_i)) \mid i = 1, \ldots, n \} \end{aligned}$$

The function symbol *modify* designates the function that modifies the attributes of C-classes. The typical model of the sentences T^0_γ is the cartesian product of the attributes specified by v_γ. Hence all the C-classes are constructed out of its attribute C-classes, if their attributes are not empty. In this sense, if a C-class has no attributes, we call it a *primitive C-class*. A C-class that is not primitive is called *compound C-class*.

In the last example of Section 3.1.2, we have shown that the set of integers is expressed as a primitive C-class. Later, we will extend this example to express the set of any C-class as a primitive C-class. This may seem a little bit strange, because it contradicts the term "primitive." It may be considered that the set of a C-class should be formalized as something complex. We use the term "primitive" meaning "structureless." In a model theoretic sense, a set C-class is structureless, the operations that are allowed to them are the standard ***union***, *intersection*, etc. There is no algebraic operation that accesses its "sub-structure."

3.2 Universal Language

Since the description of concepts is essentially local to each concept, there may be inconsistency in the name of function symbols and relation symbols. For example, a person can be concept ualized by a C-class *Person:*

$$\mathit{Person} = (\mathit{person}, \{\mathit{name}, \mathit{address}\}, v_{person}, \emptyset, \emptyset, \mathit{TRUE}).$$

On the other hand, a subconcept *Student* of *Person* may be expressed by a C-class *Student*

$$\mathit{Student} = (\mathit{student}, \{\mathit{s\_name}, \mathit{residence}\}, v_{student}, \emptyset, \emptyset, \mathit{TRUE}).$$

In this case, *s-name* and *residence* are intended to express the *name* and the *address* of the student respectively. So, in order to designate the intended equivalence of these symbols, we need a common language. We call this common language *universal language* of Γ. Later, we need the common language to define the hierarchy of the concepts. The precise definition is as follows.

3.2.1 Universal Renaming

In order to describe the correspondence of attribute names of C-class descriptions, we define the notion of *renaming* as follows. For i being 1 or 2, let L_i be a first order language made of set $\mathcal{V}_i$ of variables, set $\mathcal{F}_i$ of function symbols and set $\mathcal{R}_i$ of predicate symbols. A *renaming* α from L_1 to L_2 is a collection of injective mappings from $\mathcal{V}_1$ to $\mathcal{V}_2$, $\mathcal{F}_1$ to $\mathcal{F}_2$ and $\mathcal{R}_1$ to $\mathcal{R}_2$:

$$\alpha = (\alpha_v, \alpha_f, \alpha_r),\ \alpha_v : \mathcal{V}_1 \longrightarrow \mathcal{V}_2,\ \alpha_f : \mathcal{F}_1 \longrightarrow \mathcal{F}_2,\ \alpha_r : \mathcal{R}_1 \longrightarrow \mathcal{R}_2,$$

such that it preserves the similarity types of function symbols and predicate symbols. Namely, if a function symbol f has n arguments, $\alpha_f(f)$ also has n arguments. Similarly for predicate symbols. Note that the renaming α induces a injective mapping from L_1 to L_2.

Let L(y) be the language generated by the symbols of the description of γ. Then a language L is the *universal language* of I', if there exists a set IV of renamings such that:

$$N = \{\alpha_\gamma \mid \gamma \in \Gamma\}$$

$$\forall\gamma \in \Gamma\ \ \alpha_\gamma : L(\gamma) \rightarrow L.$$

In a practical case, we may require that the symbols of the same intended meaning will be mapped to the same symbol in the universal language. In the above example,

$$\alpha_{person}(name) = \alpha_{student}(s\_name),$$

$$\alpha_{person}(address) = \alpha_{student}(residence).$$

However these are meta-conditions. Theoretically the morphisms N determines the semantics of symbols. If we have

$$\alpha_{person}(name) = \alpha_{student}(residence),$$

it means that the 'name' of 'person' has the same semantics as 'residence' of, 'student', although it is different from the common meaning of the words "name" and "residence." The set N of renamings is called *universal renaming* of Γ.

3.2.2 Local Renaming

In the actual programming, it is difficult to describe the global semantic equality from the beginning. We can only specify the semantic equality locally, i.e. we only provide the renaming between the description languages of C-classes. In the above example, we may provide the renaming $\alpha_{Student,Person}$ from $L(Student)$ to $L(Person)$. When we have provided renaming between the description languages of individual concepts. we expect that there exists a universal renaming, which is compatible with those renamings. Before considering the existence, we introduce the conditions that those locally defined renamings should satisfy.

Let G be a subset of Γ x Γ, and let J be the set of injective renamings among $L(\gamma)$'s, such that

$$J = \{\alpha_{\gamma_1,\gamma_2} \mid \alpha_{\gamma_1,\gamma_2} : L(\gamma_1) \rightarrow L(\gamma_2)\ (\gamma_1,\gamma_2) \in G\},$$

We call (G, J) *as* the *semantic local renaming* of Γ if the following conditions are satisfied.

1. Transitivity

$$\alpha_{\gamma,\gamma'}, \alpha_{\gamma',\gamma''} \in J \Rightarrow (\alpha_{\gamma,\gamma''} \in J\ \ \wedge\ \ \alpha_{\gamma,\gamma''} = \alpha_{\gamma',\gamma''} \circ \alpha_{\gamma,\gamma'}),$$

where $\circ$ is the composition of mappings.

2. Route Independence

$$(\gamma,\gamma_1), (\gamma,\gamma_2), (\gamma_1,\gamma'), (\gamma_2,\gamma') \in G \Rightarrow \alpha_{\gamma_1,\gamma'} \circ \alpha_{\gamma,\gamma_1} = \alpha_{\gamma_2,\gamma'} \circ \alpha_{\gamma,\gamma_2}.$$

3. Acyclicity The binary relation G has no cycle.

The first condition expresses the global semantic compatibility of the morphisms. If a symbol s is semantically equivalent to a symbol s' and s' is equivalent to s'', then s should be equivalent to s'' by the transitive rule of equivalence relation. The second condition designates the consistency of the inherited attributes. The third condition describes the relevant structure of a hierarchy.

Note that we can eliminate the first condition. In fact, the second condition guarantees that we can estend (G, J) to another semantic local renaming (G', J') so that G' is transitive.

3.2.3 Existence of Universal Language

If we have a local renaming, there exists a universal language and universal renaming such that the universal renaming is compatible with the given local renaming, under a certain condition. Let us define a partial order $\preceq_G$ on Γ by the binary relation G.

$$\gamma \preceq_G \gamma' \overset{def}{\Leftrightarrow} (\gamma,\gamma') \in G.$$

Theorem 1 *Let Γ be a set of concepts, and let (G, J) be a semantic local renaming of Γ.* If *G is at most countably infinite, and Γ has the finite minimal elements with respect to $\preceq_G$, then there exists a universal language L and the universal renaming N of Γ to L, such thut*

$$\alpha_{\gamma,\gamma'} \in J \Rightarrow \alpha_\gamma = \alpha'_\gamma \circ \alpha_{\gamma,\gamma'},$$

where

$$N = \{\alpha_\gamma \mid \gamma \in \Gamma\}.$$

3.3 Fundamental Operator on C-Classes

In order to construct complex C-classes out of given C-classes, we define several operations on C-classes. These operators are some abstraction of the mental process of human beings to create new concepts out of esisting concepts. These fundamental operators correspond to the fundamental operators for data, algebras. In fact, the fundamental operators on data algebras will provide the models of the fundamental operators on C-classes.

3.3.1 Aggregation

For given C-classes, we can create a new C-class by introducing a C-class name, attribute names that correspond to given C-classes, a set of sentences that specifies the structure similar to a Cartesian product such that the attribute names are designating projections. Let $\vec{\gamma}$ be a sequence of C-classes,

$$\vec{\gamma} = (\gamma_1, \gamma_2, \ldots, \gamma_n),$$

and let Φ be a sequence of symbols with the same length as $\vec{\gamma}$,

$$\Phi = (f_1, f_2, \ldots, f_n).$$

We express each component C-class in $\vec{\gamma}$ by:

$$\gamma_i = (n_i,\ \Phi_i,\ v_i,\ T_i,\ \Delta_i,\ R_i) \quad (i = 1, \ldots, n).$$

Then the aggregation $\Pi(n_\Pi,\ \vec{\gamma},\ \Phi)$ of $\vec{\gamma}$ is defined as follows.

$$\Pi(n_\Pi, \vec{\gamma},\ \Phi) = (n_\Pi,\ \Phi,\ v_\Pi,\ T_\Pi,\ \Delta_\Pi, R_\Pi)$$

- The name n_Π of the aggregation is the symbol that is compatible with other C-classes. Namely, the symbol never appears as the name of other C-class.
- The symbols in Φ are the attribute names of the aggregated C-class $\Pi(n_\Pi,\ \vec{\gamma},\ \Phi)$.
- The attribute value v_Π is the mapping from the components of Φ to the set Γ of C-classes, such that
$$1 \leq \mathrm{Vi} \leq 72,\ v_\Pi(\ f_i) = \gamma_i.$$
- The structural sentences T_Π is similar to T^0_γ for a C-class γ with non-empty attributes.
  $$\text{Z-I} = \{\forall x\ \forall y\ f_i(modify(x, f_i, y)) = y \mid i = 1 \ldots n\} \cup$$
  $$\{\forall x\ \mathtt{T}(f_i(x), v_\Pi(f_i)) \mid \mathrm{i} = 1 \ldots n\}.$$
  The symbol modify is the function symbol for the modifier of attribute values.
- The auxiliary sentences may be any definition of new function symbols and relation symbols that simplify the description.
- Each component of the aggregation should satisfy the restrictions that are imposed on the attribute value C-classes. The restriction predicate R_Π is defined by:
$$R_\Pi(x) = \bigwedge_{i=1}^{n} R_i(f_i(x)).$$

The aggregation of C-classes has a model that corresponds to the aggregation of data algebras, which was defined in section 2.3.1. This will be discussed later.

3.3.2 Recursive Aggregation

Let G be a directed graph with a set of C-class names V as nodes and labeled edges E. Let U be a collection of nodes in V, such that there is no incoming edge. Further let W be a subset of V that contains U,

$$U \subseteq W \subseteq V.$$

We assume that for elements of W, C-classes are given. We denote an element of E as $(n,\ m,\ g)$, which designates the edge from n to m with label g. Let Φ be a set of symbols that has one to one correspondence with V,

$$\Phi = \{f_v \mid v \in V\}.$$

The recursive aggregation $\hat{\Pi}(n_{\hat{\Pi}},\ \mathrm{G},\ \Phi)$ with respect to G, Φ and W is defined as follows.

$$\hat{\Pi}_W(n_{\hat{\Pi}}, G, \Phi) = (n_{\hat{\Pi}}, \Phi, v_{\hat{\Pi}}, T_{\hat{\Pi}}, \Delta_{\hat{\Pi}}, R_{\hat{\Pi}}).$$

- The symbol $n_{\hat{\Pi}}$ is a new C-class name.
- The symbols Φ are the attribute names.

- The attribute values are provided by the one to one correspondence of Φ and V.

$$\forall v \in V,\ v_{\widehat{\Pi}}(f_v) = v.$$

- The structural sentences express the nested structure defined by G. Let V be $(v_1, \ldots, v_k)$; we consider V as a sequence.

$$\begin{aligned} T_{\widehat{\Pi}} = \ & \{\forall x_1 \ldots \forall x_k\, f_{v_i}(cons_{\widehat{\Pi}}(x_1, \ldots, x_k)) = x_i \mid i = 1, \ldots, k\} \ \bigcup \\ & \{\forall x \mathrm{T}(f_v(x), v) \mid v \in V\} \ \bigcup \\ & \{\forall x \mathrm{T}(g(f_v(x)), u) \mid (v, u, g) \in E\} \end{aligned}$$

- The auxiliary sentences include recursive definitions of restriction predicates for component C-classes.

$$\Delta_{\widehat{\Pi}} = \bigcup_{v \in V-W} \{\forall x\, R_v(x) \Leftrightarrow ((x = \nu_\perp) \vee (\bigwedge_{(v,u,g)\in E} R_u(g(x))))\}$$

where the $\nu_\perp$ is intended to designate the null value in universal algebra. For v in $V - W$, the "v'th" component of $\widehat{\Pi}(G, \Phi)$ is a. C-class with recursive structure.

- The restriction predicate designates that each component should satisfy its own restriction predicate,

$$R_{\widehat{\Pi}}(x) \equiv \bigwedge_{v \in V} R_v(f_v(x)).$$

3.3.3 Abstraction

Let γ be a C-class

$$\gamma = (n_\gamma, \Phi_\gamma, v_\gamma, T_\gamma, \Delta_\gamma, R_\gamma).$$

and let Ψ be a subset of Φ_γ:

$$\Psi = \{g_1, \ldots, g_m\} \subset \Phi_\gamma.$$

The abstraction $\Upsilon(n_\Upsilon, \gamma, \Psi)$ of γ with respect to Ψ is defined as:

$$\Upsilon(n_\Upsilon, \gamma, \Psi) = (n_\Upsilon, \Psi, v_\Upsilon, T_\Upsilon, \Delta_\Upsilon, R_\Upsilon).$$

The definition of n_Υ, T_Υ, and Δ_Υ are similar to those of aggregation.

- n_Υ is a symbol, which designates the name of $\Upsilon(n_\Upsilon, \gamma, \Psi)$.
- Ψ is the set of symbol that designates the attributes of the new C-class.
- The attribute values are the same as those of γ,

$$\forall g \in \Psi\ v_\Upsilon(g) = v_\gamma(g).$$

- T_Υ is the structural sentence defined as follows.

$$\begin{aligned} T_\Upsilon = \ & \{\forall x\ \forall y\ g_i(modify(2, g_i, y)) = y \mid i = 1 \ldots m\} \ \cup \\ & \{\forall x\ \mathrm{T}(g_i(x), v_\gamma(g_i)) \mid i = 1 \ldots m\}. \end{aligned}$$

- The restriction relation R_Υ is defined as:

$$R_\Upsilon(x) \equiv (\exists y\ \mathrm{T}(y, n_\gamma) \wedge R_\gamma(y) \wedge (\bigwedge_{g \in \Psi} (g(y) = g(x))).$$

3.3.4 Restriction

The restriction operator replaces the restriction formula of a C-class by the conjunction of the original restriction formula and an unary predicate [10]. For a C-class γ,

$$\gamma = (n_\gamma, \Phi_\gamma, v_\gamma, T_\gamma, \Delta_\gamma, R_\gamma),$$

the restriction of γ by an unary relation S is

$$\Theta(n_\Theta, \gamma, S) = (n_\Theta, \Phi_\gamma, v_\gamma, T_\gamma, \Delta_\gamma, R_\gamma \wedge S).$$

3.3.5 Set Construction

For a C-class γ

$$\gamma = (n_\gamma, \Phi_\gamma, v_\gamma, T_\gamma, \Delta_\gamma, R_\gamma),$$

the set of γ is defined as follows. This definition is an generalization of the example discussed for Set-of *_Integer* before. The relation symbols T and In have the same meaning as in the example of *Set-Of -Integer.*

$$Set(n_{Set}, \gamma) = (n_{Set}, \emptyset, \perp, T_{Set}, \Delta_{Set(\gamma)}, R_{Set(\gamma)}),$$

where

$$R_{Set(\gamma)}(x) \equiv (\forall y \text{ In}(y, x) \Rightarrow \text{T}(y, n_\gamma)).$$

The structural sentences of $Set(n_{Set}, \gamma)$ are just the theory T_{set} of set for any C-class γ. The auxiliary sentences Δ_{Set} may be defined arbitrarily to meet the appropriate description of C-classes. Although R_{Set} says nothing about the cardinality of the set, we assume that the cardinality is finite. More precisely, we only consider finite sets as the model of the set C-class $Set(n_{Set}, \gamma)$. Combining Set operation with restriction operation, we get a more general set of C-classes. More specifically, subsets of the set $Set(n_{Set}, \gamma)$ of a C-class γ will be expressed by applying a restriction operator to $Set(n_{Set}, \gamma)$.

3.3.6 Categorization

Once we get the notion of the set construction of a C-class, we can categorize the elements of the set by concerning some attributes. In the categorization, we ignore the other attributes that are not interested. We obtain *a* set of set of' *a concept* by taking a categorization. We define the *categorization operator* as follows. Let γ be a C-class, and let the *interested* attributes Ψ be a subset of the attributes Φ_γ,

$$\gamma = (n_\gamma, \Phi_\gamma, v_\gamma, T_\gamma, \Delta_\gamma, R_\gamma), \quad \Psi \subseteq \Phi_\gamma.$$

The categorization $\Omega(n_\Omega, \gamma, \Psi)$ of the C-class γ with respect Ψ is:

$$\Omega(n_\Omega, \gamma, \Psi) = (n_\Omega, \emptyset, \perp, T_{set}, \Delta_\Omega, R_\Omega),$$

where

$$R_\Omega(x) \equiv (\forall y \forall u \forall v \text{In}(y, x) \wedge \text{In}(u, y) \wedge \text{In}(v, y) \Rightarrow T(u, n_\gamma) \wedge T(v, n_\gamma) \wedge (\textstyle\bigwedge_{f \in \Psi} (f(u) = f(v)))).$$

[10] We assume that the free variable of these formulas are the same

3.3.7 Generated C-Classes

We can consider the closure by the fundamental operators on C-classes in the same manner as data algebras. The *universal family* of C-classes is the set of C-classes that is closed under fundamental operators. And the *universal closure* of C-classes is the minimum universal family that contains the C-classes.

3.4 Hierarchy of C-Classes

To formalize the hierarchy of concepts, we introduce a partial order among C-classes. We assume that *C-classes are described in a universal language.* If concepts are precisely espressetl in the real conceptual world, we can express the hierarchy of concepts by referring to only attributes. Namely a concept has more attributes than its superconcept. Thus we can express the conceptual hierarchy by inclusion of attributes. Roughly speaking, we can formalize it as follows. Let, c,c' be concepts, and let the attributes Φ_c, $\Phi_{c'}$ be the attributes of c, c' respectively. Then c is the subconcept of c' if and only if

$$\Phi_c \supset \Phi_{c'}.$$

However as we discuss in Chapter 4, our conceptualization is incomplete. Hence we cannot specify the hierarchy only by its attributes. We need to specify the hierarchy explicitly by introducing an order in the concepts. So we introduce an artificial partial order $\prec_n$ on the names of C-classes. Let n_1 ,n_2 be the name of C-classes γ_1,γ_2 respectively. We say n_1 is a *subname* of n_2 if

$$n_1 \preceq_n n_2.$$

We assume that the type matching predicate T that is introduced in Section 3.3.5 satisfies the following condition,

$$\forall n_1 \forall n_2 \; n_1 \preceq n_2 \Rightarrow (\forall x \; \mathrm{T}(x, n_1) \Rightarrow \mathrm{T}(x, n_2)\;)$$

We include above sentence as a part of our theory. With this name hierarchy, we introduce a hierarchy among C-classes.

Let γ_1, γ_2 be C-classes,

$$\gamma_i = (n_i, \; \Phi_i, \; v_i, \; T_i, \; \Delta_i, \; R_i) \; (\mathrm{i} = 1,2).$$

Then γ_1 is a *subclass* of γ_2

$$\gamma_1 \preceq \gamma_2,$$

if the following holds.

$$n_1 \preceq_n n_2, \; T_1 \models T_2,$$

$$\Phi_2 \subseteq \Phi_1, \; \forall f \in \Phi_2, \; c(v_1(f)) \preceq c(v_2(f)),$$

$$_{\mathrm{I}} = \; \forall x \; R_{\Upsilon(\gamma_1,\Phi_2)}(x) \Rightarrow R_2(x),$$

where $c(v_i(f))$ designates the C-class with name $v_i(f)$ $(i = 1,2)$, and

$$R_{\Upsilon(\gamma_1,\Phi_2)}(x) \equiv \exists y T(y, n_1) \wedge R_1(y) \wedge \bigwedge_{g \in \Phi_2} (g(y) = g(x)).$$

Since each C-class has a unique name, we could have defined the hierarchy only by the name hierarchy. However, as we discussed above, the name hierarchy is a compromise for our incomplete conceptualization. Therefore it is natural to reflect the effect of attributes in the definition of C-class hierarchy as much as possible. Thus the attributes of C-classes play the major role in determining the hierarchy of C-classes.

We should note that we can have the most general C-class in the following way. First we assume that there is the greatest element, say *top,* in the name hierarchy. Then, the most general c-class $\gamma_\top$ is:

$$\gamma_\top = (top, \emptyset, \perp, \emptyset, \emptyset, TRUE).$$

We assume that the theory $T_=$ of equality is always implicitly included in the structural sentences for any C-class γ.

$$\begin{aligned} T_= \;&=\; \{\forall x\; x = x\} \cup \{\forall x \forall y\; x = y \Rightarrow y = x\} \\ &\cup \{\forall x \forall y \forall z\; (x = y \wedge y = z) \Rightarrow x = z\}. \end{aligned}$$

Thus if we express T_γ to be empty, it means the structure is specified only by $T_=$. Namely, it is just the structure of a set.

3.5 Conceptual Order and Fundamental Operators

The conceptual order is the realization of semantic hierarchy of concepts. There is a close relation between conceptual order and the fundamental operators, as shown in the following theorem.

Theorem **2** *Let γ, γ', $\tilde{\gamma} = \{\gamma_i\}_{i=1}^n$ and $\tilde{\gamma}' = \{\gamma_i'\}_{i=1}^n$ be C-classes. Moreover, let n and n' be new C-class names such that $n \preceq n'$.*

- *Aggregation*
  For attribute names Φ,

$$(1 \leq \forall i \leq n, \gamma_i \preceq \gamma_i') \Rightarrow (\Pi(n, \tilde{\gamma}, \Phi) \preceq \Pi(n', \tilde{\gamma}', \Phi)).$$

- *Abstraction*
  For a subset Φ of the attributes of 7,

$$n_\gamma \preceq n \Rightarrow \gamma \preceq \Upsilon(n, \gamma, \Phi)$$

- *Restriction*
  For a unary predicate S,

$$n \preceq n_\gamma \Rightarrow \Theta(n, \gamma, S) \preceq \gamma$$

- *Set Construction*

$$\gamma \preceq \gamma' \Rightarrow Set(n, \gamma) \preceq Set(n', \gamma')$$

- *Categorization*
  If the set of attributes Ψ is common in y and γ', then

$$\gamma \preceq \gamma' \Rightarrow \Omega(n, \gamma, \Psi) \preceq \Omega(n', \gamma', \Psi).$$

The proof of the theorem is easy, so it is omitted.

3.6 Generalization and Specialization

In our mental processes, we generalize several concepts by taking the common attributes of those concepts. For example, we get concept 'mammal' by generalizing 'dog', 'cat', 'monkey'. etc. On the other hand, we specify a concept as the semantic intersection of several concepts. For example, the natural number is described by the semantic intersection of integer and positive number. We formalize these mental processes using the conceptual hierarchy provided above.

Let us assume that a conceptual hierarchy $\preceq$ is given. First we introduce some notations. Let $\{\gamma_i\}_{i=1}^n$ be a set of C-classes. If the least upper bound of $\{\gamma_i\}_{i=1}^n$ with respect $\preceq$ exists. we denote it by

$$\bigvee_{i=1}^n \gamma_i.$$

If $n = 2$, we denote it by

$$\gamma_1 \vee \gamma_2.$$

Dually, the greatest lower bound of $\{\gamma_i\}_{i=1}^n$ is denoted by

$$\bigwedge_{i=1}^n \gamma_i,$$

or

$$\gamma_1 \wedge \gamma_2.$$

By definition, the operator $\vee$ and A are commutative and associative. Furthermore,

$$\bigvee_{i=1}^n \gamma_i = (\gamma_1 \vee (\gamma_2 \vee (\cdots(\gamma_{n-1} \vee \gamma_n)\cdots),$$

$$\bigwedge_{i=1}^n \gamma_i = (\gamma_1 \wedge (\gamma_2 \wedge (\cdots(\gamma_{n-1} \wedge \gamma_n)\cdots).$$

Now we define the generalization and specialization.
The *generalization* of $\{\gamma_i\}_{i=1}^n$ is defined by the least upper bound $\vee_{i=1}^n \gamma_i$. In particular the generalization of two C-classes γ and γ' is $\gamma \vee \gamma'$. As stated above, any generalization is described by the operator $\vee$. *We* call $\vee$ the *generalization operator.* The definition of the *specialization* is similar to that of the generalization. We replace $\vee$ and "least upper bound'? in the definition of generalization by A and "greatest lower bound" respectively. We call the operator $\wedge$ the *specialization operator.*

Similarly, we introduce operators $\vee, \wedge$ in the C-class names, according to the name hierarchy.

Due to theorem 2, we have the following theorem.

Theorem 3 *Let* γ, γ', $\tilde{\gamma} = \{\gamma_i\}_{i=1}^n, \tilde{\gamma}' = \{\gamma_i'\}_{i=1}^n$ *be C-classes.*

- *Aggregation*
  Let $\tilde{\gamma} \wedge \tilde{\gamma}'$ *be the sequence* $(\gamma_1 \wedge \gamma_1', \ldots, \gamma_n \wedge \gamma_n')$, *and let* $\tilde{\gamma} \vee \tilde{\gamma}'$ be $(\gamma_1 \vee \gamma_1' \ldots \gamma_n \vee \gamma_n')$.
  For a new C-class name n, n', n'', *a sequence of attribute names* Φ,

$$n \wedge n' = n'' \Rightarrow \Pi(n, \tilde{\gamma}, \Phi) \wedge \Pi(n', \tilde{\gamma}', \Phi) = \Pi(n'', \tilde{\gamma} \wedge \tilde{\gamma}', \Phi),$$

$$n \vee n' = n'' \Rightarrow \Pi(n, \tilde{\gamma}, \Phi) \vee \Pi(n', \tilde{\gamma}', \Phi) \preceq \Pi(n'', \tilde{\gamma} \vee \tilde{\gamma}', \text{(a)}.$$

- *Abstraction*
  For a common subset Φ *of attributes of* γ,γ',

$$n \wedge n' = n'' \Rightarrow \Upsilon(n,\gamma,\Phi) \wedge \Upsilon(n',\gamma',\Phi) = \Upsilon(n'',\gamma \wedge \gamma',\Phi),$$

$$n \vee n' = n'' \Rightarrow \Upsilon(n,\gamma,\Phi) \vee \Upsilon(n',\gamma',\Phi) = \Upsilon(n'',\gamma \vee \gamma',\Phi).$$

- *Restriction*
  For a unary predicate S,S',

$$n \wedge n' = n'' \Rightarrow \Theta(\gamma, S \wedge S') = \Theta(\gamma, S) \wedge \Theta(\gamma, S'),$$

$$n \vee n' = n'' \Rightarrow \Theta(\gamma, S \vee S') = \Theta(\gamma, S) \vee \Theta(\gamma, S').$$

- *Set Construction*

$$n \wedge n' = n'' \Rightarrow Set(n,\gamma) \wedge Set(n',\gamma') = Set(n'',\gamma \wedge \gamma'),$$

$$n \vee n' = n'' \Rightarrow Set(n,\gamma) \vee Set(n',\gamma') \preceq Set(n'',\gamma \vee \gamma').$$

We should note that in the previous two theorems, we always have to specify the name hierarchy to obtain a reasonable result. The name hierarchy is an artificial hierarchy and we have to assign the order in the names of C-classes so that they are compatible to the natural semantic hierarchy of concepts.

To summarize, we have introduce the notion of C-class and an order among them to formalize concepts and the semantic hierarchy of concepts. Moreover we have introduced formal operators on C-classes that provides a formalism of mental processes that produce new concepts out of existing concepts. Finally, we have provided some theorems to show that the formalism provides the natural relation between the fundamental operators and the concept hierarchy, which is one of the verifications of the correctness of the formalism.

4 Models and Instances

So far, we have discussed the notion of C-classes, which is the formalization of database schema objects. Now, we are going to discuss the actual data that will be in a database. We regard a database as an expression of the real world. Each concept in the real world is expressed by C-class defined in the previous chapter. Each occurrence of concept is expressed as an instance of C-class.

In the framework of a value-oriented model, an instance of a C-class is just an element of the data algebra that is the model of the C-class. The occurrence of a compound C-class is determined by the set of attribute values. However, as we discussed in Section 1.1, we cannot capture the real existence of the occurrence in this paradigm, because our conceptualization is always incomplete, *i.e.,* an approximation of the real concept. We need something other than attribute values to distinguish the occurrences in the real world. It is so-called *object-identity,* which will be formalized.

In this chapter, *we* first define the *value-oriented model* of C-classes. A value-oriented model of C-classes is a collection of data algebras that are specified by the C-classes. The data algebra provides the space where the structure of the real world objects are expressed. Next, we will estend the value-oriented model to *object-oriented model* by introducing the object-identity space.

4.1 Value-Oriented Model of C-Classes

Let Γ be a set of C-classes generated by fundamental operators from a set Γ_0 of the primitive C-classes, and let

$$\mathbf{D} = (\{\delta_\gamma \mid \delta_\gamma = (n_\gamma, A_,, r_\gamma), \gamma \in \Gamma\}, \preceq_n)$$

be the pair of a many-sorted data algebra with the sort S generated by Γ_0, and the name-hierarchy $\preceq_n$ of data algebras. Then D is called a *value-oriented model* of I', if the following conditions are sat isfied. Let γ be an element of Γ such that:

$$\gamma = (n_\gamma, \Phi_\gamma, v_\gamma, T_\gamma, \Delta_\gamma, R_\gamma).$$

- Primitive C-Classes
  Each primitive C-class 7 satisfies:
  - The universal algebra A, is the algebra. corresponding to a sort in S.
  - The restriction function of δ_γ is the interpretation of R,. We assume that each predicate will be interpreted as a function to 2, where 1 is regarded to be *TRUE.*
- Compound C- Classes
  For any compound C-class 7, A, is a subalgebra of $\Pi_{f\in\Phi_\gamma}\mathbf{A}_{v(f)}$. Typically, when T_γ is equal to T_γ^0, A, is isomorphic to the product algebra $\Pi_{f\in\Phi_\gamma}\mathbf{A}_{v(f)}$ itself.
  - Each function symbol f in Φ_γ is interpreted as the projection from $\prod_{f\in\Phi_\gamma}\mathbf{A}_{v(f)}$ to $\mathbf{A}_{v(f)}$.
  - The restriction function $\mathbf{r}_\gamma$ is also the interpretation of R,.

For a C-class 7 corresponding a concept, an element of the data algebra δ_γ represents an occurrence of the concept as a value. We call the element a *value instance* of 7. Furthermore, the data algebras should be compatible with the hierarchy of C-classes. Namely,

$$\forall\gamma\forall\gamma' \in \Gamma, 7 \preceq \gamma' \Rightarrow \exists\rho_{\gamma,\gamma'} : \delta_\gamma \rightarrow \delta_{\gamma'}, (\rho_{\gamma,\gamma'} \text{ is the subtype mapping from 7 to 7'}).$$

For the top C-class, we have a model $\delta_\top$ that is set theoretically isomorphic to the set of object-identities. which will be formally introduced in the nest section.

$$\delta_\top = ((\Omega, \emptyset), 1).$$

4.2 Object-Oriented Model of C-Classes

The value-oriented model of a C-class provides the base of the algebraic structure for expressing occurrences of concepts. In this section, we extend the value-oriented model by the notion of object-identity. We will introduce ***object-identity space*** to express the real existence of objects.

Let D be a value-oriented model of I' as defined in the previous section. Let $\mathbf{\Omega}$ be the pair of a set Ω with an appropriate cardinality, a collection $\mathcal{F}$ of partial functions from Ω to itself. We call $\mathbf{\Omega}$ the *object-identity space.* Further, let I be a. collection of partial functions from Ω to a data, algebras in D for each 7 in Γ. Namely,

$$\mathbf{D} = \{\delta_\gamma | \gamma \in \Gamma\},$$

$$I = \{\iota_\gamma | \iota_\gamma \colon \Omega \rightarrow \delta_\gamma, \gamma \in \Gamma\}.$$

The partial function $\imath_\gamma$ is called the *instance mapping* of 7. The domain $\partial(\gamma$j of ι_γ is called the *object instances* of 7.

Then, an object-oriented model $\mathcal{M}(\Gamma)$ of C-classes I' is a triplet

$$\mathcal{M}(\Gamma) = (\mathbf{D}, \mathbf{\Omega}, I),$$

which satisfies the following conditions.

- Let 7, 7' be in I. If $\gamma \preceq$ 7' then

  $$\partial(\gamma) \subseteq \partial(\gamma') \text{ and } \forall \omega \in \partial(\gamma)\ \imath'_\gamma(\omega) = \rho_{\gamma,\gamma'} \circ \imath_\gamma(\omega).$$

  where $\rho_{\gamma,\gamma'}$ is the subtype mapping from 7 to 7' in the value-oriented model D. This condition shows the compatibility of the hierarchies of the object-identity space and the value-oriented model D. Note that the hierarchy of C-classes in the object-identity space is expressed by the set inclusion of the domains of instance mappings.

- For each function symbol f that appears in the description of C-classes, there is a. corresponding partial function o(f) in $\mathcal{F}$.

- The mappings o(f)'s are related to the value-oriented interpretations $\nu(f)$'s via the mappings I in the following way. Let us take a function symbol f that appears in the description of C-classes, which has a. signature" $n_1 n_2 \ . \ . \ . \ n_m \rightarrow$ n, where n and n_i's are concept names of C-classes γ, 7;'s. Then we have the commutative equation:

  $$\imath \circ o(f) = u(f)\ 0\ \pi^n_{i=1}\ \imath_i,$$

  $$\begin{tikzcd} \Omega^n \arrow[r, "\pi^n_{i=1}\ \imath_i"] \arrow[d, "o(f)"'] \arrow[dr, phantom, "\circlearrowright"] & \Pi^n_{i=1}\delta_i \arrow[d, "\nu(f)"] \\ \Omega \arrow[r, dash] & \delta \end{tikzcd}$$

  where

  $$U(f) : \Pi^n_{i=1}\ \delta_i \rightarrow \delta,\ o(f) \colon \Omega^n \rightarrow \Omega,$$

  $\pi^n_{i=1}\ \iota_i$ is the product mapping of $\imath_i$ (i = 1, . . . n):

  $$\pi^n_{i=1}\imath_i((\omega_1, \ldots, \omega_n)) \overset{def}{=} (\imath_1(\omega_1), \ldots, \imath_n(\omega_n)).$$

  The data algebra. δ corresponds to the C-class 7, and ι is the instance mapping of γ, similarly data algebra, δ_i and instance mapping ι_i for γ_i (1 $\leq i \leq$ nj.

"The definition of signature is provided in [GB 85].

The above commutativity is the essence of our model. It clearly separates the "object-oriented part" and "value-oriented part". We call it *fundamental commutativity.* Further, it demonstrates the essential difference of an object-oriented data model and a value-oriented data model. The difference between object-oriented model and value-oriented model lies in the object identity. There are several features other than object-identity, which are generally considered to characterize an object-oriented model. such as complex object, inheritance, etc. However, as we will see later, the semantics of those features can be captured by the algebraic construct, such as types, aggregation operators, when we express instances as elements(values) of a data algebra.

The set of instance mappings $\{\imath_\gamma \mid \gamma \in \Gamma\}$ is called an *schema instance* of Γ.

The object-identity space $\mathbf{\Omega}$ is a *flat*[12] set with a set of partial functions. The value-oriented model D provides a structure on $\mathbf{\Omega}$, which is called *value space* of Γ. The instance mapping of a C-class expresses the correspondence between object instances and value instances.

We have a natural ordering for schema instances. Let the object-identity space $\mathbf{\Omega}$ and value-oriented model D be fixed, and let I and I' be schema instances of Γ:

$$I = \{\imath_\gamma \mid \gamma \in \Gamma\},\ I' = \{\imath'_\gamma \mid \gamma \in \Gamma\}.$$

We call the schema instance I the *schema subinstance* of I' and denote it by

$$I \preceq I',$$

if

$$\forall \gamma \in \Gamma,\ \imath'_\gamma \text{ is an extension of } \imath_\gamma.$$

This ordering is useful when we consider the schema instances of C-classes with recursive structure. Obviously, the order is a partial order. If I and I' coincide on the intersection of their domains,

$$\forall \gamma \in \Gamma, \forall x \in \partial(\imath_\gamma) \cap \partial(\imath'_\gamma),\quad \imath_\gamma(x) = \imath'_\gamma(x).$$

we call them *compatible.* It is easy to prove that *any set of compatible schema instances has the least upper bound with respect to the above order.*

4.3 Induced Mapping on Instances

In this section, we discuss how the fundamental operators on C-classes are interpreted in the object-oriented model.

The *induced fundamental operators* are the mappings that transform instance mappings to other instance mappings. For given C-classes, we can create new C-classes using fundamental operators. Accordingly, for the created C-classes, we can create instance mappings out of instance mapping of original C-classes. In this section, by the term "instance mapping", we mean a, partial function. from object-identity space to a data algebra, which *may* provide an object-oriented model. As discussed later, the induced instance mapping will not provide an object-oriented model for a certain kind of fundamental operators.

Let us assume that an object-oriented model $\mathcal{M}(\Gamma)$ of Γ is given:

$$\mathcal{M}(\Gamma) = (\mathbf{D}, \mathbf{\Omega}, I), I = \{\imath_\gamma : \Omega \longrightarrow \delta_\gamma \mid \gamma \in \Gamma\}.$$

[12]By the term flat, we mean that no element of the set has a. substructure.

We assume D and $\boldsymbol{\Omega}$ are fised. As defined before, an instance mapping $\imath_\gamma$ is a partial function from the object-identity space $\boldsymbol{\Omega}$ to the data algebra δ_γ. We denote the domain of an instance mapping $\imath_\gamma$ by $\partial(\imath_\gamma)$. Let $\imath_\gamma$ be the instance mapping of γ in I,

$$\gamma = (n_\gamma,\ \Phi_\gamma,\ v_\gamma,\ T_\gamma,\ \Delta_\gamma,\ R_\gamma).$$

- Restriction

  Let S be a unary predicate that is intended to impose a restriction on γ. The induced *restriction operator* $\widetilde{\Theta}(\cdot,$ S) is defined

$$\partial(\widetilde{\Theta}(\imath_\gamma, S)) \overset{def}{\equiv} \{\omega \in \partial(\imath_\gamma) \mid \nu(S)(\imath_\gamma(\omega)) = 1\}$$

$$\forall\omega \in \partial(\widetilde{\Theta}(\imath_\gamma)),\ \widetilde{\Theta}(\imath_\gamma, S)(w) \overset{def}{\equiv} \imath_\gamma(\omega).$$

  Intuitively, the induced restriction operator takes only instances that satisfy the predicate S. Note that the predicate symbol S is interpreted as a mapping from δ_γ to 2.

- Abstraction

  Let Ψ be a subset of Φ. The induced abstraction operator $\widetilde{\Upsilon}(\cdot,\ \Psi)$ is defined by:

$$\partial(\widetilde{\Upsilon}(\imath_\gamma, \Psi)) \overset{def}{\equiv} \partial(\imath_\gamma),$$

$$\forall\omega \in \partial(\widetilde{\Upsilon}(\imath_\gamma)),\ \widetilde{\Upsilon}(\imath_\gamma, \Psi)(\omega) \overset{def}{\equiv} P_\Psi \circ \imath_\gamma(\omega),$$

  where P_Ψ is the projection from $\Pi_{f\in\Phi}\mathbf{A}_f$ to $\Pi_{g\in\Psi}\mathbf{A}_g$.

- Aggregation

  The induced operator for aggregation is different from the above operators, because it is a constructive operator. Let 7; be a C-class and let $\imath_i$ be the instance mapping for γ_i $(i = 1, \ldots, n)$. Then the induced aggregation operator $\widetilde{\Pi}(\cdot)$ is defined as follows. The domain $\partial(\widetilde{\Pi}((\imath_1 \ldots \imath_n)))$ of the induced mapping is a new subset of Ω that has one to one correspondence to $\Pi_{i=1}^n\ \partial(\imath_i)$ with a mapping ϵ:

$$\epsilon : \partial(\Pi((\imath_1, \ldots, \imath_n)) \xrightarrow{\sim} \Pi_{i=1}^n \partial(\imath_i).$$

  Then the induced instance mapping is defined by:

$$\widetilde{\Pi}((\imath_1, \ldots, \imath_n)) \overset{def}{\equiv} (\Pi_{i=1}^n \imath_i) \circ \epsilon.$$

  There is a certain technical details, about the aggregation operator. If we have already an instance mapping $\imath$ for the aggregated C-class, we impose a condition to the invention of object identities so that the newly derived instance mapping is an extension of the esisting one.

- Recursive Aggregation

  The induced operator for a recursive aggregation is obtained by inductive limit of generated instances. More precisely, we first define an inflational operator to produce new instances. Then we take the limit of successive applications of the operator.

  Let G', V, E, U, W and Φ be the same as in Section 3.3.2. Let Γ be the set of C-classes corresponding V, and let D be the object-oriented model of I',

$$\Gamma = \{\gamma_u \mid u \in V\}.\ \mathrm{D} = \{\delta_u \mid u \in V\}.$$

Further, let $\mathcal{I}$ be the collection of all schema instances of I'. We define an operator ζ_W from the $\mathcal{I}$ to $\mathcal{I}$. Let I be in $\mathcal{I}$,

$$I = \{\imath_u : \Omega \rightarrow \delta_u \mid u \in V\}.$$

Then

$$\zeta_W(I) \overset{def}{=} I \vee \xi_W(I)$$

where $\vee$ designates the least upper bound with respect to the schema instance ordering defined at the end of section 4.2. The instance mappings $\xi_W(I)$ is defined as the minimal schema instance that is compatible with I such that it satisfies the following conditions.

$$\begin{array}{lll} \forall v \in W, & \xi_W(I)_v = \imath_v, & \\ \forall v \in V - W, & \forall u \forall g \text{ s.t. } (v, u, g) \in E, & \pi_g(\Im(\xi_W(I)_v)) \supseteq \Im(\imath_u). \end{array}$$

where $\Im(\xi_W(I)_v))$ and $\Im(\imath_u)$ are codomain of $\xi_W(I)_v$ and $\Im(\imath_u)$ respectively, and π_g is the project ion correspondin g the edge (v, u, g). Note that $\xi_W(I)$ may not be unique [13]. For a given schema instance I, we construct a monotone increasing schema instance sequence $\{I_n\}_{n=0}^{\infty}$ by applying ζ_W successively.

$$I_0 = I, \quad I_{n+1} \overset{def}{=} \zeta_W(I_n).$$

Since $\{I_n\}_{n=0}^{\infty}$ forms a compatible set of schema instances, we can obtain the inductive limit I_∞ as the least upper bound of the set. Then we define the induced schema instance $\widehat{\Pi}(G, \Phi. \text{ IV})$ as $I,$:

$$V = \{v_1, v_2, \ldots, v_m\}, I_\infty = \{\hat{\imath}_i : \Omega \rightarrow \delta_{v_i} \mid 1 \leq i \leq m\}.$$

$$\widehat{\Pi}(G, \Phi, \mathrm{W}) = \widehat{\Pi}((\hat{\imath}_1, \ldots, \hat{\imath}_m)).$$

By definition, the instance mappings for the C-classes in W will not change with ζ_W. We call W the set of *stable C-classes.* If W is equal to V, the recursive aggregation reduces to the original aggregation defined above.

- For a. set construction, we can naturally induce an instance mapping. The induced instance mapping describes the instances with all the possible finite sets of original instances. More precisely, let $\imath$ be an instance mapping of a C-class γ.

  $$\imath : \Omega \rightarrow \delta_\gamma.$$

  Then induced mapping $\jmath$ by set construction is a minimal instance mapping such that its codomain includes all finite sets generated by the codomain of $\imath$.

  $$\Im(\jmath) \supset \{\{x_1, x_2, \ldots, x_n\} \mid x_i \in \Im(\imath)\ (1 \leq i \leq n), n = 0, 1, 2, \ldots\}.$$

  The induced mapping is not unique. If the C-class $Set(n, \gamma)$ has non-null instance mapping $\imath_{set}$ from the beginning, we construct the instance mapping $\jmath$ so that $\jmath$ is the extension of $\imath_{set}$.

[13] Actually, ξ_W is a multi-valued function. However, we consider it as an ordinary function by taking one of the values. The existence of ξ_W can be easily proven using the fundamental commutativity.

- Categorization

  The induced mapping for the categorization is obtained by the composition of induced mappings of set construction, aggregation, and restriction operators, according to the definition of the categorization.

We should notice that the induced instance mapping may not be unique for (generalized) aggregation, and set construction. This is due to the fact that these operators require object-identity invention [AK 89].

Furthermore, we can introduce operators on instances that correspond to generalization/specialization operators.

Let γ_i be a C-class,

$$\gamma_i = (n_i,\ \Phi_i,\ v_i,\ T_i,\ \Delta_i,\ R_i)\ (i\ =\ 1,2),$$

and let $(\Pi_{f\in\Phi_i}\mathbf{A}_{i,f},\ \mathbf{r}_i)$ be the data algebra corresponding to γ_i. Further, let $\imath_i$ be an instance mapping of γ_i, and let P_i be the projection from $\Pi_{f\in\Phi_i}\mathbf{A}_{i,f}$ to $\Pi_{f\in\Phi_1\cap\Phi_2}\mathbf{A}_{i,f}$ (i = 1, 2). If

$$\forall f\in\Phi_1\cap\Phi_2\ \mathbf{A}_{1,f}=\mathbf{A}_{2,f} \text{ and } P_1\circ\imath_1 = P_2\circ\imath_2 \text{ on } \partial(\imath_1)\cap\partial(\imath_2),$$

the induced generalization and specialization of $\imath_1$ and $\imath_2$ are defined as follows.

- Generalization

  The induced generalization operator $\tilde{\vee}$ is defined as:

  - if the intersection of Φ_1 and Φ_2 is not empty,

$$\partial(\imath_1\tilde{\vee}\imath_2) \overset{def}{\equiv} \partial(\imath_1)\cup\partial(\imath_2),$$

$$\forall\omega\in\partial(\imath_i)\ (\imath_1\tilde{\vee}\imath_2)(\omega) \overset{def}{\equiv} P_i(\imath_i(\omega))\ \ (i\ =\ 1,2).$$

  - if the intersection of Φ_1 and Φ_2 is empty, the domain of $\imath_1\tilde{\vee}\imath_2$ is the same as above, and

$$\begin{array}{lccll}(\imath_1\tilde{\vee}\imath_2): & \Omega & \rightarrow & \delta_{\mathrm{T}} & = ((\Omega,\emptyset),1)\\ & \omega & \longmapsto & \omega & (\text{ inclusion mapping }).\end{array}$$

- Specialization

  For the specialization operator on C-classes, we have the following induced specialization operator $\tilde{\wedge}$. The operator $\tilde{\wedge}$ is defined as:

$$\partial(\imath_1\tilde{\wedge}\imath_2) \overset{def}{\equiv} \partial(\imath_1)\cap\partial(\imath_2),$$

$$\forall\omega\in\partial(\imath_1\tilde{\wedge}\imath_2),\ (\imath_1\tilde{\wedge}\imath_2)(\omega)\in\prod_{f\in\Phi_1\cup\Phi_2}\mathbf{A}_f,$$

$$\Pi_i\circ(\imath_1\tilde{\wedge}\imath_2) = \imath_i\ (i\ =\ 1,2).$$

Although we can derive new instances by induced operators, we should note that these instances are just, possible candidate instances in our model. However, in intuitive sense, if a C-class is derived by the fundamental operator other than aggregation or set construction, the instance mapping should be obtained by the induced operators. We should note that our object-oriented model is fairly general. Hence we would get *a* variety of "actual models" according to the way of providing instance mappings. To provide instance mappings by the induced mappings of the fundamental operators is a. canonical way of obtaining an object-oriented model.

5 Database Design

5.1 Entity C-Classes and Abstract C-Classes

In our object-oriented model of C-classes, there can be more than one object-identity corresponding to one element of data algebra. Because our conceptualization is incomplete, we cannot characterize the real existence of objects by their attribute values. However, in order to provide a representation, we should assume that the existence can be described by attribute values for certain concepts at least in a closed domain of the real world. This is a matter of knowledgebase design.

Hence it is important to analyze in which case a C-class should be characterized by its attribute values, or more generally, in which case the object instances are equivalent to the value instances. Namely, we should consider when we should require the instance mapping of C-class to be injective. In this section, we consider two kinds of C-classes that the instance mapping will be injective. One is the algebraic C-class, the other is the logical C-class. Further we claim that even the instance mapping of a logical C-class has the inherent possibility of not being injective, because our knowledge representation is always incomplete.

First, we introduce and discuss the algebraic C-classes. Let us consider the concept *string* for example. What are the instances of *string*? It depends on the context how we consider the concept. We can say that every string appearing in the real world can be an instance of C-class *String*. Consider the following same sentences.

- "string" is an instance of *String*.
- "string" is an instance of *String*.

The string "string" in the *first sentence* is an instance of *String* which is different from the instance "string" in the *second sentence*. However, we often need to abstract the real occurrences of *String* and regard the many instances as a same object. This is exactly what the value-oriented model of C-class *String* is intended to be. The universal algebra $\mathbf{A}_{String}$ is the abstraction of real occurrences of strings with abstracted functions such as *length*, *conctrtenute*. The algebraic model $\mathbf{A}_{string}$ is virtual and doesn't exist in the real world. However, we want to treat the virtual model, such as the algebra $\mathbf{A}_{String}$, as if it existed in the real world. In other words, we want to allow the conceptual existence of the abstract objects. So we introduce a. category of C-classes whose instances are virtually the same as the domain of an algebra in the value-oriented model. Namely, the instance mapping is injective. We call such C-classes *algebraic C-classes.* An algebraic C-class is a. kind of "literal."

Other than algebraic C-classes, there is another kind of C-classes that instance mapping should be injective. It is the C-class derived from a logical relation. We can express a. n-ary logical relation by a C-class with n attributes. Since an occurrence of logical relation is nothing but an element of a subset of the Cartesian product of doma.ins(object-identities), it is exactly characterized by its attribute values. We call such C-classes *logical* C-classes. The notion of logical C-classes will be discussed in detail with an example later in this section.

Note that the notion of algebraic C-classes and logical C-classes are *not* determined by object-oriented models. Rather, it is required in the meta level. In other words, it is a design issue of knowledge representation whether we require a C-class to be an algebraic or logical C-class. We call a C-class an *abstract C-class*, if we impose a restriction that its instance mapping is injective.

The abstract c-classes strictly fit into the value-oriented data model. If all the C-classes are abstract C-classes, any object-oriented model is essentially the same as a value-oriented model.

We call the remaining C-classes *entity C-classes*, whose instance mappings are not, intended to be injective. The entity C-classes are the representation of the "existing objects" in the real world. In a practical design of knowledgebase, the physical objects and events are espressed as entity C-classes. This design issue will be discussed in the later section. For example, 'person','animal','company', 'meeting' and 'order' are entity C-classes. Note that the "esisting objects" should not necessarily be physical objects nor events. It can be some abstract object, which is still an expression of the existence of "something" in the real world. Basically, anything that can be noun will be an entity C-class. Hence, even 'friendship', 'love' can be entity C-classes. Actually, the author presumes that the nominalization in the mental process of human being is essentially the same as creating an entity C-class. The identity of an entity C-class is characterized by its *object-identity*.

We emphasize again that the notion of abstract C-class and entity C-class is not determined by its model. The instance mapping of an entity C-class may be injcctive with some particular object-oriented model. It is a meta level requirement, i.e. design level requirement.

It is controversial whether we should express a logical relation as a C-class. Alternatively, we can introduce the notion of logical relation as another construct of our theory. There are two reasons why we express logical relations as C-classes.

- It may be the case that an occurrence of logical relation will be converted to an existing object by a certain meta operation, which will be discussed in the rest of this section. So, it is more convenient to express logical relations as C-classes, because the meta operation can be expressed as just a mapping from a C-class to another C-class.
- It is better to have only C-classes as the basic construct of the model so that we can treat the knowledge representation in a simple and homogeneous way.

In the rest of this section, we will provide the intensive consideration to the meaning of entity C-classes and logical C-classes. Especially, we will discuss the meta operation that converts a logical C-class to an entity C-class.

A logical C-class is a compound C-class that we make up to express a logical relation of the real world objects.

Let us consider a concept *Person* with attributes, *name, loving*. Further, let $\imath$ be the instance mapping of *Person* and $\partial(\imath)$ be the domain of $\imath$. The at tribute value $loving(\omega)$ designates the people that ω loves.

$$Person\ = (person,\ (name.\ loving),\ v_{Person},\ T_{Person},\ TRUE)$$

$$v_{Person}(name)\ =\ String.\quad v_{Person}(loving) = Set\_of\ \_Person.$$

For esample,

$$w,\ w' \in \partial(\imath),\ name(w)\ =\ \text{“}John\text{”},\ name(\omega')\ =\ \text{“}Mary\text{”}.\ \mathrm{In}(\omega',\ loving(\omega))$$

means that the person w named "John" loves the person w' named "Mary."

A C-class $\heartsuit$ will be a logical C-class with two attributes 'loves' and 'loved' pointing persons;

$$\heartsuit = (affection, \{loves, loved\}, v_{\heartsuit}, T_{\heartsuit}.\ R_{\heartsuit}),$$

where

$$v_{\heartsuit}(loves)\ =\ person, v_{\heartsuit}(loved)\ =\ person,$$

$$R_{\heartsuit}(x) \equiv \mathrm{In}(loved(x), loving(loves(x))),$$

and the structural sentence $T_{\heartsuit}$ is the one that is similar to T^0_{γ} for a C-class γ with non-empty attributes. It is important that the existence of the 'affection' is derived from the attributes(state) of the persons. In this case, it is derived from the attributes of the loving person. The restriction form $R_{\heartsuit}$ is not only *restriction* but also the *definition of* the C-class $\heartsuit$. Namely, the existence of instance is exactly specified by $R_{\heartsuit}$. Generally, the occurrence of a logical C-class γ is specified by the the restriction predicate R_{γ}. Thus the identity of a logical C-class should be determined completely by its attribute values. The occurrences of a logical C-class should be the same if and only if their attribute values are the same. Thus one might say that a logical C-class can be dealt with by the value-oriented paradigm. However it is not so simple.

We should notice that even a logical C-class is an approximation of the real world. In the above example, we specified the C-class *affection* with the predicate $R_{\heartsuit}$. If the predicate completely specifies an "affection", the attribute values will determine the equivalence of instances. However, it does not. 'John loved 'Mary' yesterday, i.e. the predicate $R_{\heartsuit}$ held for 'John' yesterday, but it doesn't hold today. Even in such case, we can still think "yesterday's love of John for Mary." The instance of concept acquired an object identity. The reason is that the specification by the predicate $R_{\heartsuit}$ had lacked temporal information. If it had included the temporal attribute, we could have expressed the "yesterday's love" only by attribute values. Therefore, due to the incompleteness of our representation, even a logical C-class may end up as an entity C-class. Hence we introduce a meta operation $\mathcal{N}$ that converts a logical C-class to an entity C-class. We call $\mathcal{N}$ a *nominalization operator*. The nominalization operator corresponds to the mental process of putting a name to a chunk of information that we acquired.

As discussed above, every C-class may be inherently an entity C-class. However, in order to organize the knowledge representation. we should impose a condition that certain C-classes are to be abstract C-classes, as discussed in the nest section.

5.2 The Concept Model

In this section, we introduce *concept model* for database design, and discuss its semantics.

5.2.1 Design Process

First, we discuss the design of knowledge representation. As we mentioned in the previous section, even **an** instance of logical relation would be an instance with object-identity. However, when we develop a knowledge representation, we have to *assume* some of the C-classes should be abstract C-classes. For example, when we register a new instance of C-class in the knowledgebase, we have to know whether the instance is already stored or not. As we discussed, we can only believe that we can distinguish the instances by our representation. This is a. matter of correctness of knowledge representation. Hence, when we design a knowledge representation using C-classes, it is the main issue what C-classes we should regard as the basic abstract C-classes.

The design process will consists of the following steps.

1. Provide algebraic C-classes, such as *Integer. String*, *Set*, *Sequence*. Further we provide primitive functions and predicates. For example, $\{ +, -, >, \ldots \}$ for *Integer*, $\{$*union*, *intersection*, In$\}$ for Set.

2. Choose real world concepts that provide the basis of our knowledge representation and express them by C-classes. We introduce as many attributes as possible to those C-classes, so that we can assume that their inst ances are fully specified by at tribute values, i.e., the instance mapping is injective. We call such C-classes *base C-classes.* For example, a concept *person* would be expressed by *a* base C-class *Real-Person.* We assign as many attributes as possible so that we can distinguish individual persons. (The concepts, such as *employee, student* can be espressed by C-classes derived from *Real-Person* by abstraction operator, because we don't need all the attributes of *Real-Person* to express an employee or a student .)

   We should note that basic C-classes are inherently entity C-classes, although we regard them as abstract C-classes. In fact, when we view the knowledge representation through a perspective different from the original design or when we add a new C-class into the schema, a base C-class may become an entity C-class. In such a case, we have to modify the schema by adding new attributes to the base C-class, in order to keep up our requirement that the C-class should be an abstract C-class.

   The guidelines of selecting base C-classes are as follows.

   - *Physical objects* should be base C-classes. For instance, person, car, location. etc. Social organizations, such as company, may be considered as physical objects, because they consists of physical objects, such as employee, office, factory, etc.
   - *Events* should be base C-classes. For instance, meeting, accident, order form of parts, etc.

3. Analyze the relation of base C-classes and check that every necessary logical relation among base C-classes can be expressed by the attributes of base C-classes. We add new attributes, if necessary. The point is that *all information should be included in the attributes* of *base C-classes.* If so, we can express any information by the C-classes derived by the fundamental operators from base C-classes. Hence, the integrity constraints of knowledgebase will be completely described by the restriction predicates of base C-classes. Thus in order to maintain the consistency, we only have to maintain that of base C-classes.

   For example, when we consider a C-class *Person* and a C-class Car, there may be a logical relation $OwnerCar$. We express them with attributes *owns* of *Person* and *owner* of Car. The attribute *owns* designates the belongings of a person, and the attribute *owner* designates the owner of a car. Then we will express the $OwnerCar$ relation by a logical C-class with attributes *{owner. car},* and the restriction predicate $R_{ownercar}$:

$$OwnerCar = (ownercar, \{owner, car\}, v_{ownercar}, T^0_\gamma, \emptyset, R_{ownercar}),$$

$$v_{ownercar}(owner) = person, \; v_{ownercar} = object,$$

$$R_{ownercar}(x) \; 3 \; \mathrm{In}(car(x), owns(owner(x))),$$

   where T^0_γ is the same as in section 3.1.3. The restriction predicate means that the car *car(x)* is one of the belongings of the person *owner(x).*

   It is an important requirement that we can construct every logical relation by attributes and primitive functions and predicates of algebraic C-classes $\mathcal{A}$ and base C-classes $\mathcal{B}$. If so, we can construct any logical relation through fundamental operators from $\mathcal{A}$ and $\mathcal{B}$. Hence it will allow us to provide the semantics of those logical C-classes using induced instance mappings. We will discuss it in the next section.

4. We define appropriate "view" C-classes using fundamental operators. Logical C-classes will be defined by the (generalized) aggregation operator, while entity C-classes will be defined by abstractions and restriction operators.

5.2.2 The Concept Model and Its Semantics

A *concept model* $\mathcal{M}$ of a knowledgebase is a tuple consisting of C-classes of three kinds together with a C-class hierarchy $\preceq$.

$$\mathcal{M} = ((A, \mathcal{B}, \mathcal{D}), \preceq).$$

A is the set of *algebraic C-classes* such *as Integer, String,* etc. $\mathcal{B}$ is the set of *base C-classes.* $\mathcal{D}$ is the set of all *derivable C-classes*, which can be derived by a. finite application of the fundamental operators from $A \cup \mathcal{B}$. We should note that the union of A, $\mathcal{B}$ and $\mathcal{D}$ forms the universal closure of the union of A and $\mathcal{B}$.

The semantics of the model is as follows. Let Γ be a finite subset of the union of A, $\mathcal{B}$, $\mathcal{D}$, such that for each C-class in I', the C-classes that are the attribute values of γ is also in Γ:

$$\gamma = (n_\gamma, \Phi_\gamma, v_\gamma, T_\gamma, \Delta_\gamma, R_\gamma),$$

$$\forall f \in \Phi_\gamma \; v_\gamma(f) \in \Gamma.$$

We call such a set of C-classes closed *set of C-classes.*

The semantics of the concept model is provided by an object-oriented model $(\mathbf{D}, \mathbf{\Omega}, I)$ of the C-classes I' with the following conditions for I. Let δ_γ be a data algebra in D that is the model of γ in Γ.

- The instance mapping of a C-class γ in A is injective and surjective partial function from Ω to δ_γ.
- The instance mapping of a C-class γ in $\mathcal{D}$ is obtained by induced instance mapping of the fundamental operators that define the C-class. For a recursive aggregation, we require that the base C-classes are always treated as stable C-classes. We will consider this ind riced mapping in detail in the next section.
- The instance mappings of base C-classes espress the instances that are existing in the real world. The instance mapping of a C-class γ in $\mathcal{B}$ would be intended to be injective by the knowledgebase designer. However, we don't impose the restriction as part of the formal semantics. If the instance mapping happens to become not being injective, the schema, of the knowledgebase should be altered. It is a matter of maintenance of schema.. Note that a base C-class may be defined with fundamental operator from other base C-classes and algebraic C-classes. However, the instance mapping is not derived by induced instance mapping. The instances will be created by update operations of the user.

As we discuss in Appendix A, one of the characteristics of this model is homogeneous representation of query. There is no distinction between those three kinds of C-classes for users, so long as query is concerned. A user doesn't have to consider which C-class corresponds to the data. stored in the knowledgebase. Each C-class would be automatically bound to a set of instances by the system. The homogeneity of C-classes will bring a clear semantics of view update, which will be discussed in Section A.2.4.

5.2.3 Two Kinds of Predicates

If the derived C-classes are recursively defined, their instance mappings will not be always determined nor exist. In this section, we consider this matter further.

First we extend the graph we discussed in the definition of generalized aggregation. We allow the labels of edges to be operator expressions that express the other fundamental operators. For example,

$$Person = (person, (name, height, father), v_{Person}, T_{Person}, \emptyset, TRUE),$$

$$v_{Person}(name) = string, v_{Person}(height) = Integer,$$

$$v_{Person}(father) = person,$$

$$Tall\ Person = \Theta(tallperson, Person, R_{TallPerson}),$$

$$R_{TallPerson}(x) = (height(x) \geq 6\ (ft.)).$$

The graph will be:

$$\begin{aligned} V &= \{ person, tallperson, string, integer), \\ E &= \{ (person, string, name), (person, integer, height), \\ &\quad (person,, person, father), \\ &\quad (tallperson, person. \Theta(tallperson, \bullet\ . R_{TallPerson})\} \end{aligned}$$

We can define a function ζ from schema instances to themselves in a similar way as in section 4.3. The difference lies in deriving the new instance mapping of the C-classes V_d that are derived by fundamental operators other than aggregation.

$$\forall v \in W, \xi(I)_v = \imath_v,$$

$$\forall v \in V - W - V_d, \forall u, g s.t.(v, u, g) \in E, \pi_g(\Im(\xi_W(I)_v)) \supseteq \Im(\imath_u),$$

$$\forall v \in V_d - W, (v, u, expr) \in E\ \xi(I)_v = \langle \text{the induced instance mapping by } expr \rangle.$$

$$\zeta(I)_v = \begin{cases} \imath_v \vee \xi(I)_v & (\text{if } v \in V - V_d), \\ \xi(I)_v. & (\text{otherwise}). \end{cases}$$

As shown later in this section, this ξ will produce a non-sense instance mappings for a certain class of restriction operators.

Next, we introduce a meta function symbol *getinstances* in the language that designates all the instances of a C-class. For a C-class γ and its name n_γ, $getinstances(n_\gamma)$ designates the set of object-identities in $\partial(\imath_\gamma)$. The set $getinstances(n_\gamma)$ can be regarded as an instance of $Set(\ n_{set}, \gamma\ )$. For example, we consider a base C-class Man and a derived C-class $Richestman$.

$$Man = (\ man, \{name, wealth,. . .\}, v_{Man}, T_{Man}, \emptyset, TRUE),$$

$$Richestman = (richestman, \{name, wealth\}. v_{Richestman}, T_{Richestman}, \emptyset, R_{Richestman}),$$

$$v_{Man}(name) = v_{Richestman}(name) = string,$$

$$v_{Man}(\ wealth) = v_{Richestman}(\ wealth) = integer,$$

$$R_{Richestman}(x) \equiv (\forall y\ In(y, getinstances(man)) \Rightarrow wealth(\ x) \geq wealth(y)\ ).$$

The *getinstances* cause the interpretation of predicates to be dependent on the instance mappings. Hence, it may not be a consistent instance mapping to some C-class definition. For example, we can express an inconsistently defined C-class:

$$WrongNumber = \Theta(wrongnumber, \Pi(n001, (Integer), (value)), R_{WrongNumber}),$$

$$v_{WrongNumber}(value) = integer,$$

$$R_{WrongNumber}(x) \equiv (\forall y \; In(y, getinstances(wrongnumber)) \Rightarrow x \neq y).$$

We should note that this kind of inconsistency comes from semantics of instances. It is different from a relevant, inconsistent restriction predicate. such as

$$R(x) \equiv (P(x) \wedge \neg P(x)).$$

The operator ζ defined above gives us the wrong answer in this case. Let us assume I is the initial schema instance.

$$I_{integer} : \Omega \longrightarrow \mathbf{Z}(\text{onto, one to one}),$$

$$\partial(I_{integer}) = \{\omega_1, \omega_2, \ldots, \},$$

$$I_{wrongnumber} = \perp.$$

where $\perp$ is the null mapping:

$$\perp : \Omega \longrightarrow \mathbf{Z} \; (\partial(\perp) = \emptyset).$$

Then, by definition, we have:

$$\xi(I)_{integer} = I_{integer}, \; \xi(I)_{wrongnumber} = I_{integer},$$

$$\zeta(I)_{integer} = I_{integer}, \; \zeta(I)_{wrongnumber} = I_{integer},$$

$$\xi(\zeta(I))_{integer} = I_{integer}, \; \xi(\zeta(I))_{wrongnumber} = \perp,$$

$$\zeta(\zeta(I))_{integer} = I_{integer}, \; \zeta(\zeta(I))_{wrongnumber} = \perp.$$

In general,

$$\xi(I_n)_{wrongnumber} = \begin{cases} I_{integer} & (\text{if } n \text{ is even}) \\ \perp & (\text{if } n \text{ is odd}), \end{cases}$$

where I_n designates $\zeta^n(I)$. Thus, we cannot have the inductive limit of $\{I_n\}_{n=1}^{\infty}$. The problem comes from the fact that $R_{wrongnumber}$ depends on its own instance mapping. More specifically, the variable y is universally quantified on the domain of the instance mapping. So, $\xi(I_n)$ "oscillates" between $I_{integer}$ and $\perp$. The induced mapping of $WrongNumber$ doesn't provide an object-oriented model.

According to this observation, we introduce a class of predicates.

First,, we introduce the following syntax sugar to simplify the notation.

$$(\forall x : n_\gamma \; \phi) \stackrel{def}{\equiv} \forall x(\, In(x, getinstances(\, n_\gamma)) \Rightarrow \phi),$$

$$(\exists x : n_\gamma \, \phi) stackreldef{\equiv}\exists x(\, In(x, getinstances(n_\gamma))A \; \phi).$$

Then the above example is denoted by:

$$R_{WrongNumber}(x) \equiv (\forall y : wrongnumber \; x \neq y).$$

We call the expressions $x : n_\gamma$ *a explicitly typed variables,* and $\forall(\exists)\ x : n_\gamma$ a *explicitly typed quantifier.* For any first order formula, we can move each explicitly typed quantifier to left side of the expression, in the same manner as ordinary quantifiers. For example,

$$\forall x : n\,(T(x) \Rightarrow \exists y : m\, P(x,y))$$

becomes

$$\forall x : n\, \exists y : m\ \neg T(x) \vee P(x,y).$$

We call the first order form *a normally quantified form,* if each explicitly typed quantifier is placed at the left side of the expression.

If a first order form has a normally quantified form with only existential explicitly typed quantifiers, we say that it is of *type 2.* A general first order form is called *type 1.*

Theorem 4 *If every restriction predicate is* of type *2, then for each schema instance I, the operator ζ defined in this section has a fix point I_∞ such that I is a subinstance* of I_∞.

We can prove that the restriction operator is monotone increasing with respect to the order among instances. So, we can prove ζ is monotone increasing. Hence, there exists an inductive limit by the fact mentioned at the end of Section 4.2.

In this section, we have introduce a forma.l semantics for the concept model. The semantics is espressed by a. fixed point of ζ-operator. The fixed point of i-operator doesn't exist in some case. We can consider such a concept model as inconsistent. Theorem 4 shows that some class of concept model is consistent in the sense that there exists a fixed point of ζ[14].

6 Expressibility of Concept Model

In this chapter, we consider the expressibility of our model by simulating other models.

6.1 Relational Model Semantics

The relational model can be simulated by a. concept model. Since we will show that datalog semantics can be simulated by a concept model in the next section, we can derive this result as an easy corollary. However, we can prove it directly. In this section we provide only the sketch of the proof.

We express relations as compound C-classes. For esample, a. relation *Person(name, address)* will be expressed by a C-class:

$$Person = (person,\ \{name,\ address),\ v_{person},\ T^0_{person},\ \emptyset,\ TRUE),$$

$$v_{person}(name) = v_{person}(address) = String.$$

The relational operators are simulated by induced operators of the fundamental operators.

selection	$\longleftrightarrow$	*restriction*
projection	$\longleftrightarrow$	composition of categorization and abstraction
product	$\longleftrightarrow$	*aggregation*

[14]There is a trivial case that the fixed point, always exists. If there are no recursive aggregation involved in the definition of the derived C-classes, then the concept model is consistent, i.e., the ζ-operator has a fixed point. In fact, for an initial schema instance I, $\zeta(I)$ is the fixed point.

Furthermore we have a natural interpretation for the natural join operator. It is expressed by the specialization operator. Let R and S be relations and γ_R and γ_S be the corresponding C-classes. Then

$$R \bowtie S \longleftrightarrow \gamma_R \wedge \gamma_S.$$

6.2 Datalog Semantics

In this section, we show that the semantics of datalog can be simulated by the concept model. First we discuss how to convert datalog rules to C-class definitions. We assume that algebraic C-classes such as *Integer, String* are provided from beginning. We introduce some terminology. A *simple rule* is a rule with the body consisting of one literal. If a rule is not simple, we call it a *complex rule.* We call predicates such as =, < , *restrictive predicates* and literals such as X < *1 restrictive literals.* We also assume that all rules are *rectified*[15]. Moreover, we assume:

- There is no predicate symbol that is used with different arity. For example, we don't have the rules such as:

```
p(X, Y) :- x = Y.
p(X) :- x > 0.
```

We convert. rules into the forms that will be easily transformed to C-class definitions in the following way.

1. If the predicate symbols of facts appear as the heads of rules, we add new rules so that they never appear in rules. For example, the rules:

```
p(a).
p(X) :- q(X) .
will become
p1(a).
p(X) :- p1(X) .
p(X) :- q(X).
```

2. If there is a variable that is shared by more than one negated literal, and doesn't appear in positive literals, we rename the variable so that it is not shared by negated literals. For example,

```
p(X) : - ¬q(X,Y) & ¬s(X,Y) & t(X).
will become
p(X) : - ¬q(X,Y) & ¬s( X. Z) & t(X).
```

3. We convert the rules by adding equality literals so that the non-restrictive literals do not share any variable.

```
p(x) :- q(X,Y) & Y = 1.
will become
```

[15][UL 88] Chapter 3.

```
p(X) :- q(Z,Y) & Z = X & Y = 1.
```

4. If a negated non-restrictive literal shares variables with restrictive literals, we seperate them by introducing "intermediate" equalities. For example,

   $p(X) :- \neg q(Y,Z) \,\&\, Y = x \,\&\, z = 1.$
   will become
   $p(X) :- \neg q(Y', Z') \,\&\, Y' = Y \,\&\, Z' = z \,\&\, Y = x \,\&\, z = 1.$

   We call the expressions like Y' = Y, Z' = Z in the above example the *intermediate literals* and distinguish them from restrictive literals by using the equality symbol $\overset{\circ}{=}$ instead of $=$. So the second rule in the above example is expressed by:

   $p(X) :- \neg q(Y', Z') \,\&\, Y' \overset{\circ}{=} Y \,\&\, Z' \overset{\circ}{=} z \,\&\, Y = x \,\&\, z = 1.$

For the rules after the above conversion, we assign C-classes as follows.

1. For each non-restrictive literal symbol, we assign a. C-class (taking the predicate symbol as its name).
2. For each argument of a non-restrictive literal, we assign numbered literal names as the attribute names. For example, a literal p(X, Y, Z) has attributes, $p1$,$p2$,$p3$. $p1$ corresponds to X, $p2$ to Y and $p3$ to Z. Let us denote the correspondence by α. In the above esample,

$$\alpha(X) = p1, \alpha(Y) = p2, \alpha(Z) = p3.$$

3. For each variable, we assign a C-class name as follows. We express the assignment by a mapping τ.
   - If a variable appears in a restrictive literal, we assign the name of an algebraic C-class according to the literal. For example, if we have $X = 1$. we get

$$\tau(X) = integer^{16}.$$

   - Otherwise. we assign the most generic C-class name *top*:

$$\tau(X) = top.$$

   We should remember that we assumed the existence of the most generic C-class top γ_T in the C-class hierarchy.

4. For each attribute, we assign a C-class name in the the following way. We determine the values of attribute value function v_p for each literal symbol p. In the above example,

$$v_p(p1) = \tau(X), v_p(p2) = \tau(Y), v_p(p3) = \tau(Z).$$

5. We convert bodies of rules to first order forms with explicitly typed quantifiers. We describe the way of conversion with examples. We express the conversion with a mapping ϕ.

[16]If X is paired wit.11 different, types(C-classes) by equalities, we assign the least upper bound of those C-classes to the variable X. For example, if 'X = 1' and 'X = "ab"', we assign *top* to X.

- restrictive literal
  We convert the variables as shown in the following example.

$$\phi(\mathtt{X} = 1) = \begin{cases} \alpha(\mathtt{X})(self) = 1 & \text{(if X is in the head of the rule)} \\ \alpha(\mathtt{X})(x_p) = 1 & \text{(if X is in a non-restrictive literal } p(...)\text{)} \\ x = 1 & \text{(otherwise)} \end{cases}$$

  where self will be the free variable in the restriction predicate of the restriction operator. The variable x_p designates the instance of C-class p.
- intermediate literal
  Let X' $\overset{\circ}{=}$ X be an intermediate literal, where X' is in a negated non-restrictive[17] literal and X is in a restrictive literal. The variable X will be converted in the same way as in the restrictive literal. We denote it by $\phi(\mathtt{X})$. The variable X' is converted to $\alpha(\mathtt{X}')(x_p)$ where p is the literal symbol that contains X. So X' $\overset{\circ}{=}$ X will be converted to $\alpha(\mathtt{X}')(x_p) = \phi(\mathtt{X})$.
- non-negated non-restrictive literal
  We assume that non-restrictive literals are placed on the left side of restrictive literals in the bodies of rules.

$$\phi(p(\mathtt{X})) = \exists x_p : p.$$

- negated non-restrictive literal

$$\phi(\neg p(\mathtt{X}, \mathtt{Y})) = \forall x_p : p.$$

After the above conversion, we add explicitly typed quantifiers for the variables that appear only in the restrictive literals.

- If the variable X appears only in a negated literal, we add $\forall x : \tau(\mathtt{X})$.
- Otherwise, we add $\exists x : \tau(\mathtt{X})$.

We arrange the existential quantifiers left side of the universal quantifiers. Next we collect the intermediate literals for each negated non-restrictive literal and take the disjunction of negation of the literals. For example,

$$p(\mathtt{X}, \mathtt{Y}) :- \neg q(\mathtt{W}, \mathtt{V})\ \&\ s(\mathtt{A}, \mathtt{B})\ \&\ \mathrm{X} = \mathrm{W}\ \&\ \mathrm{V} = \mathrm{B}\ \&\ \mathrm{A} = \mathrm{Y}\ \&\ \mathrm{B} = \mathrm{C}.$$

will become

$$p(\mathtt{X}, \mathtt{Y}) :- \neg q(\mathtt{W}', \mathtt{V}')\ \&\ s(\mathtt{A}, \mathtt{B})\ \&\ \mathrm{W}' = \mathrm{W}\ \&\ \mathrm{V}' = \mathrm{V}\ \&\ \mathrm{X} = \mathrm{W}\ \&\ \mathrm{V} = \mathrm{B}\ \&\ \mathrm{A} = \mathrm{Y}\ \&\ \mathrm{B} = \mathrm{C}.$$

Then its body will be transformed to:

$$\begin{aligned} \exists c : top\, \exists u : top\, \exists v : top\, \exists x_s : s\, \forall x_q : q \quad & ((\neg(q1(x_q) = u) \vee \neg(q2(x_q) = v)) \wedge \\ & (p1(self) = u \wedge s2(x_s) = v \wedge s1(x_s) = p2(self) \wedge s2(x_s) = c)). \end{aligned}$$

Finally, we convert rules to C-class definitions.

[17]More precisely, we should say non-restrictive and non-intermediate literal. However we use the term "non-restrictive literal" in this sense.

- For rules with head literal $p(X_1, X_2, \ldots, X_n)$ with bodies B_i ($1 \leq i \leq$ m),

$$p(X_1, X_2, \ldots, X_n) :- B_1$$

$$p(X_1, X_2, \ldots, X_n) :- B_2$$

$$p(X_1, X_2, \ldots, X_n) :- B_m$$

the C-class γ_p is defined as:

$$\gamma_p = \Theta(p, \Pi(`C001', (p1, \ldots, pn), (\tau(X_1), \ldots, \tau(X_n)), \bigvee_{i=1}^{m} \phi(B_i)).$$

- For facts, we assign each predicate symbol of facts a C-class. For example, for the following fac t s,
  f(1,"abc").
  f("a","bc").

  we have

$$\gamma_f = \Pi(f, (f1, f2), (top, string)).$$

We regard that all the C-classes are abstract C-classes. We construct a. concept model with:

- Algebraic C-classes, such as Integer, String, are given.
- Base C-classes are those obtained from facts.
- The rest of the C-classes are regarded as derived C-classes.

If we provide the instance mappings for a.11 the base C-classes according to ground facts, we can get the datalog semantics as the least fised point of <-operator. If there is no negated subgoal, ζ is monotone increasing, because the restriction predicates are type 2. Thus ζ has the inductive limit as its fixed point. If we have stratification, we can get the least fixed point of ζ by the algorithm described in Chapter 3 of [UL 88].

6.3 IQL Semantics

We show that our model can espress the semantics of Abiteboul and Kanellakis' IQL-model.

In the following discussion, the meaning of notations is the same as theirs, unless it. is explicitly mentioned. We have the sets of relation names R, class names P, attributes A. and constants D, and object identities 0. A given schema (R, P, T) is converted by introducing new class names P' so that each type expressions appearing concerning T is depth 1. For each class name p in P $\cup$ P',

$$\mathbf{T}(p) = D \mid p' \mid [A_1{:}p_1, \ldots, A_n{:}p_n] \mid \{p'\} \mid (p_1 \vee p_2) \mid (p_1 \wedge p_2),$$

where $p', p_1, p_2, \ldots, p_n$ are in P $\cup$ P'.

For example, if we have type assignment,

$$\mathrm{T}(person) = [name{:}[first{:}string, last{:}string], age{:}integer],$$

we convert it as:

$$T(person) = [name{:}person\_name,\ age{:}integer],$$

$$T(person\text{-}name) = [first{:}string,\ last{:}string].$$

Another example is that:

$$T(set\text{-}of\_rational) = \{[den{:}integer,\ num{:}integer]\}$$

will be converted to:

$$T(set\text{-}of\text{-}rational) = (rational),$$

$$T(rational) = [den{:}integer, num{:}integer].$$

Next we change the syntax of literals in Abiteboul-Kanerakis' paper. We convert each literal expression $t_1(t_2)$ to $\mathtt{In}(t_2, t_1)$, where t_1 is of type $\{t_2\}$. Furthermore, for a type assignment τ for variables in rules, we introduce new C-class names so that the value of τ is always a class name. For example, if we have a rule:

$$p([A_1{:}X, A_2{:}Y]) \longleftarrow q(X).\ r(Y).$$

and type assignment for variables:

$$\tau(X) = [den{:}integer,\ num{:}integer],\ \tau(Y) = integer,$$

we convert the type assignment by introducing a class name *rational1* and a new type assignment :

$$r(X) = rational,\ r(Y) = integer,$$

$$T(rational) = [den{:}integer,\ num{:}integer].$$

Furthermore, for each type expression that appears in a rule, we assign a new class name, which will be also included in P'. We introduce a new class $p1$

$$T(p1) = [A_1{:}\tau(X),\ A_2{:}\tau(Y)].$$

Finally, we convert the rule using new type assignment and class names, together with newly introduced variables. For example, the above rule will be:

$$p(Z) \longleftarrow q(X), r(Y), A_1(Z) = X, A_2(Z) = Y,$$

$$\tau(Z) = p1, T(p1) = [A_1{:}\tau(X),\ A_2{:}\tau(Y)].$$

We extended the syntax by interpreting A_1 and A_2 as a function symbol.

After this conversion, we have:

- class names $P \cup P'$,
- the estended type assignment T' for classes and τ' for variables, (Note that we can assume that each rule has the disjoint set of variables),
- new rules with only variables as the argument of relation symbols $R \cup \{\mathtt{In}\}$.

Now, we create the C-classes according to an extended schema and modified rules in the following way. First. we convert the schema into C-classes.

1. T(p) = p'
   We replace each class name p by p' in the schema expression and rules
2. $\mathrm{T}(p) = \{p'\}$
   We use set construction.
   $$\gamma_p = Set(p, \gamma_{p'}).$$
3. T(p) = $p_1 \vee p_2$
   $$\gamma_p = \gamma_{p_1} \vee \gamma_{p_2}.$$
4. $\mathrm{T}(p) = p_1 \wedge p_2$
   $$\gamma_p = \gamma_{p_1} \wedge \gamma_{p_2}.$$
5. $[\mathtt{A}_1{:}p_1, \ldots \mathtt{A}_m{:}p_m]$
   We use recursive aggregation to define γ_p's.
   $$\gamma_{dummy} = \widehat{\Pi}(dummy, G, \Phi),\ G = (V, E),$$
   $$V = \{p \in \mathrm{P} \cup \mathrm{P}' \mid p \text{ appears in the aggregation expression.}\},$$
   $$E = \{(p, p', \mathtt{A}_k) \mid \mathrm{T}(p) = [\mathtt{A}_1{:}p_1, \ldots, \mathtt{A}_k{:}p', \ldots].\}$$
   Φ is any set of symbols that has one to one correspondence with V.

Nest we convert the rules into C-class by the same way as we convert datalog rules. The only difference is that we may have a functional expression, such as $\mathtt{A}_1(X)$, as argument of equality. We can convert such an expression naturally to a first order formula. In the above example, the rule:

$$\mathtt{p(Z)} \longleftarrow \mathtt{q(X), r(Y), A_1(Z) = X, A_2(Z) = Y.}$$

would be converted into

$$\gamma_p = \Theta(p, p1, (\exists x_q{:}q\ \exists y_r : \mathtt{A}_1(self) = q1(x_q) \wedge \mathtt{A}_2(self) = r1(y_r))).$$

For given IQL program Γ(S, S_{in}, S_{out}), we convert the schema S and the rules in the above way and get C-classes. Then we define a concept model with

- The C-class γ_D for the constants D is the only algebraic C-class.
- The C-classes that correspond to the initial ground fact are base C-classes, as in the case of datalog program.
- The remaining C-classes are derived C-classes.

Then the programs inflational fixed point will be provided by a fixed point of the <-operator. Note that providing the instance of a schema in the IQL model is the same as providing a set of ground facts.

6.4 IRIS Semantics

In this section, we briefly show that most of the semantics of IRIS system [FS 89] can be espressed by a concept model. We provide only a sketch of simulating the IRIS semantics by the concept model.

Up to now, we assumed that the algebras that appear in the value-oriented model C-classes are partial-valued algebras. In ordered to capture the semantics of IRIS system, we assume that they are multi-valued algebras. We need no change in our theory, because we can replace the partial functions in our discussion by multi-valued functions, because the multi-valued functions and sets form a category as we suggested in Section 2.1.

We formalize the semantics of IRIS system without foreign functions. First we assign algebraic C-classes to its literals, such as integers, strings. Second we assign base C-classes to its objects. Finally, we describe the functions by first order sentences and add them to the auxiliary sentences of C-classes. Then the object-oriented model of these C-classes provides the semantics of IRIS system. Actually, the semantics is expressed exactly by the object-identity space of the object-oriented model.

7 Future Work

There are several issues for future work.

- Schema Evolution
  As suggested in Chapter 5, object-identity plays an essential role of schema maintenance. It may provide the formal guideline for schema evolution. For example, when a new concept (schema object) is added to schema, the existing concepts should be altered so that base concepts will stay being abstract concepts.

- Complex Values
  We demonstrated that complex value has an inherent disadvantage concerning maintenance of consistency of a. knowledgebase, because it cannot incorporate with object-sharing. However, it has a strong advantage in providing structured data that a programmer can easily handle, as discussed in [LR 89]. Hence we should introduce the formalism that can provide the structured data without sacrificing object-sharing. The author presumes that it would be attained by introducing "local concept." Namely, the language provides the construct for defining concepts that are local to a concept. A programmer can provide the access method to the local concepts so that the instance of local concept and its attributes can be shared from outside. We should note that this will bring no change in the semantics of object-identity. Any object-identity is inherently global, because knowledge is global. The object-identity of a local concept is realized in the "global" object-identity space, as well as that of global concept. The construct of the local concepts will be introduced for programming convenience.

- Implementation of Concept Model
  Recently, a prototype system of Concept, Model has implemented the model as a language. The prototype system is written in 12,000 lines of Common Lisp code. The system checks the integrity constraints automatically. The actual session performed on the prototype system is shown in Appendix C.

  There are several technical issues. such as type checking consistency maintenance and object-binding, which will be discussed in the nest. report.

8 Conclusion

We have presented a formalism that expresses the clear semantics of object-identity and the essential distinction of the value-oriented model and the object-oriented model. In order to express the value-oriented semantics, we have introduced the notion of data algebras. The semantics of object-oriented model is expressed by the combination of the object-identity representa tion and the value-orieuted representation.

Moreover, the formalism has incorporated the logical database model *into* the object-oriented model by expressing logical relations as classes.

We should emphasize that our model provides the full-advantage of object-sharing using object-identities, when it is applied to a practical system. Yet, it also provides the structured algebraic semantics.

The concept model based on the formalism has been proposed, which provides the formal guidelines on knowledgebase design. The concept model is an attempt to represent the existing objects in the real norld as faithfully as possible. Namely, the instances of base C-classes are strictly corresponding to the esisting objects. Then the abstraction of those objects is expressed by derived C-classes. The model provides a way of expressing and maintaining the integrity constraints easily.

Acknowledgment

The authors would like to express thanks to Professor Stefano Ceri for the discussion. The authors also thank to Peter Ii. Rathmann and Dallan W. Quass for their comments on the draft of this report, and to Marianne Siroker for the careful reading and the correction of English.

This research has been supported by contract N00039-84-C-0211 of DARPA. The views and conclusions in this document are those of authors and should not be interpreted as representative of the official policies, either expressed or implied, of DARPA or U.S. Government.

A Database Operation

So far, we have discussed the schema representation of database. In this chapter, we will describe the database operations, query and update.

A.1 Query

The semantics of query is simple for the concept model $\mathcal{M}$,

$$M = ((\mathcal{A},\mathcal{B},\mathcal{D}),\preceq).$$

A query is basically to get instance mapping of a concept γ in $\mathcal{A} \cup \mathcal{B} \cup \mathcal{D}$. We take the minimal closed set I' of concepts that contains γ in the union of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{D}$. Then we obtain a fixed point of ζ operator for Γ. As discussed in the previous chapter, for a certain concept, there may not exist the fixed point.

A.2 Update

The update is to modify the object-model of concepts, i.e., to modify the instance mappings. We assume that the value-oriented model and object-identity space are fixed. Further, we assume that any update is obtained by composing the following three operations.

A.2.1 Insertion

Basically, the insertion can be done to base concepts. Or when we insert an instance to a derived concept, it should be transformed to the insertion of a base concept. Thus we cannot insert to a. derived concept obtained by the constructive aggregation. On the other hand, we can insert an instance to a concept derived by the restriction operator. If we allow "null-valued" attributes, we can insert an instance to a concept derived by the abstraction operator.

The procedure for insertion is as follows.

Create a new object-identity, say ω.

2. Register the values of attributes, say Φ, of ω. More specifically, modify the interpretation $o(f)$'s of f in Φ. If the value(object-identity) doesn't exist, we create and insert it recursively.

3. Check the integrity constraints. If the constraints are not satisfied, then undo the operat ion. (Signal error.)

A.2.2 Deletion

Theoretically, we don't allow the deletion of object-identity, because object-identity is something that expresses the real existing object. For example, even if a person dies, the fact of the existence of the person cannot be eliminated from our knowledge. However, in a practical system, we may eliminate the object-identity if the object-identity is no longer referred to by the objects of our interest. This operation is performed by a kind of garbage collection.

A.2.3 Modification

When we modify an attribute value of an instance w, we change the interpretation of the function symbol, say f, that corresponds to the attribute. More specifically, we change the value of $o(f)(w)$. The modification should be compatible with the value-oriented model. If the object-identity for the new value of $o(f)(\omega)$is not in the knowledge base, we create a new object-identity with the same procedure for insertion.

A.2.4 View Update

Since we have a homogeneous representation of concepts, we can update the knowledgebase through derived concepts, whenever it is possible. More precisely, if we can specify a unique object-identity(instance) to be deleted or modified, then we can delete the instance or modify the attribute value of the instance. When we insert an object through view concept, if we can verify the object doesn't exist as an instance of base concept, we can convert the insertion operation to the insertion of the object-identity to a base concept.

To summarize, if the update can be mapped to a. unique update at base concept level, then it can be performed. There is a typical case when update through derived concept can be done safely. *If a derived concept is derived from $\mathcal{A}$ and $\mathcal{B}$ only through abstractions and restrictions, then the deletion and modification can be mapped to a unique update of the base concept,* because the induced instance mapping of the concept derived by abstraction and restriction has a smaller domain than that of instance mapping of the base concept.

B Methods, Overloading, Encapsulation

The methods and encapsulation can be formalized simply by using functions with subtype matching. We should note that we don't distinguish the type and class in our model. A *C*-class plays the role of type. In other words, each type will be assigned to only one class. Since we have C-class hierarchy, there is no semantic reduction even without the distinction of class and type. In this chapter, we use the term *type* instead of *C-class*, when we use a C-class as type.

B.1 Method by Function

All methods are defined as a function with strong type checking. A method of a C-class γ is defined by a **binary** function. One argument type for the function is γ, the other is the type for the message. Note that we allow a. multiple function definition in the following sense. For each function name, we can have the multiple definition, so long as the tuple of the argument types of the function is different. The tuples of the argument types are ordered by the product order derived from the C-class hierarchy. Hence, the compiler will try to pick up the most specific function definition according to the argument types. For example, if we have the espression

$$(f x_1 \ldots x_n)$$

[18], we pick up the function definition of f with the minimal type tuple that matches the types of $(x_1 \ . \ . \ . \ x_n)$. We require the minimal type t uple to be unique. In a. practical system, if there exists more than one minimal type tuple, then the compiler will signal an error.

[18] We use a lisp-like notation of function.

B.2 Overloading

The overloading of methods is naturally attained, because the most specific function definition is taken for a particular pair of type and message.

B.3 Encapsulation by Subtype Matching

The encapsulation is realized by the C-class hierarchy. Let us assume that C-class γ_1 is a. super class of γ_2.

$$\gamma_1 \preceq \gamma_2, \quad \gamma_i = (n_i, \Phi_i, v_i, T_i, \Delta_i, R_i) \ (\ i = 1, 2 \ )$$

The attributes that are proper for γ_1 cannot be accessed from γ_2. In other words, the argument to the function in $\Phi_1 - \Phi_2$ should be an instance of a subclass of γ_1. Note that we include a C-class itself to its subclass.

By type casting, we can easily provide a way to define a method of γ_2 that can access the attributes proper to γ_1. For example, let $(\star \ . \ \cdot)$ be the type casting function. If a variable x has a type γ_2, and γ_1 is the subtype of γ_2, then $(\star \ \gamma_1 \ x)$ has type γ_1. Then we can define a function like in the following example.

(defunction `fun1` (x : γ_2, m: $\gamma_{\mathtt{m}}$)
    (`f` ($\star$ γ_1 `x`))...),
where `f` is the function with argument type γ_1.

B.4 Application to Database Security

The encapsulation can be used for database security. In this section, we describe the rough sketch of the idea.. First a user is provided with a set of C-classes that he/she can access. More specifically, the type names that the user can use for the type declaration is restricted in the access language. So, we could say that each user has the different access language. Let us denote the set of accessible types for a user u by d(u). We call it *access domain*. The restriction of accessible C-classes is used as follows, for example. When we want to restrict a user to access only instances of a C-class that satisfy a certain condition, it can be easily realized by allowing the user to access only to the C-class derived from the C-class by a restriction operator.

A user who can access only some higher level of types is not able to access the attributes proper to the subtypes of them without a type casting function. Hence, we can impose a. protection by restricting the use of the type casting function. The protection mechanism is quite simple. A user is provided with a set of types that can be used as the destination type of type casting function. In the above example, each user has the restriction for the first argument of $(\star \cdot \cdot)$. Let 'P(u) be the set of types that a user u is allowed to use in type casting function. The set $\mathcal{P}(u)$ is called the *access range* of a user u. Let us call u a *supervising user* of *type* γ if $\mathcal{P}(u)$ contains γ . If a. user u needs a method that should access the attributes of a. type that are not in the access range nor in the access domain, u should ask a supervising user of the type for defining the function. Then the defined function is shipped to u. Each user u. has the set of *given functions* $\mathcal{F}(u)$ that he/she can use other than functions of his/her own definition. The shipped function is added to $\mathcal{F}(u)$. Therefore, the protect ion is completely characterized by the triplet $(\mathcal{A}(u), \mathcal{P}(u), \mathcal{F}(u))$ of access domain, access range and given functions. We call it *access privilege*. Furthermore, we could introduce a relevant order to designate the strength of access privilege. Let us denote the set of all access privileges by $\mathcal{P}$. Let α, β be in P.

$$\alpha = (\mathcal{A}_\alpha, \mathcal{P}_\alpha, \mathcal{F}_\alpha), \beta = (\mathcal{A}_\alpha, \mathcal{P}_\beta, \mathcal{F}_\beta).$$

The access privilege ck is stronger than β, if

$$\mathcal{A}_\alpha \supseteq \mathcal{A}_\beta \text{ and } \mathcal{P}_\alpha \supseteq \mathcal{P}_\beta \text{ and } \mathcal{F}_\alpha \supseteq \mathcal{F}_\beta.$$

Moreover, we can estend the notion of access privilege by assigning protection with each of database operations, such as read and write, insert and delete. Let C be the categories of operations. The extended access privilege Π is the collection of mapping from C to $\mathcal{P}$.

We can manage the access of user by II together with the access hierarchy provided by the partial order of access privileges.

For example, it is natural to require that write protection is tighter than read protection. Then, it is expressed by:

$$\forall f \in \text{IT}, f(\text{'write'}) \preceq f(\text{'rend'}).$$

We can also introduce the order in Π. For f, g in II, g has stronger access power than f if

$$\forall c \in \mathcal{C}, f(c) \preceq g(c).$$

Then users can be organized by Π with this order. For example, a manager would have stronger access power than his staff members with this order.

C ADL Sample Session

As we mentioned earlier, the implementation of the formalism in this report is in progress. It is realized as a data description language called **ADL**(Algebraic Data Language). Currently, the system is made of 12,000 lines of Common Lisp Code. It has the following features.

1. CLOS-like Functional Language
   It has CLOS-like functional language with strong type checking for hierarchical types, i.e., it allows subtypes. We can attach a restriction predicate to each class to express the integrity constraints.

2. Lazy Evaluation of Object-binding
   The binding of instances to each class will be delayed until necessary. Moreover, the update of instances are performed according to the local logs of classes. The dependency of classes, such as "what update of which class will affects which class" is checked at compile time. Since the object-binding is done according to the local update logs, the update cost is smaller and we can perform a necessary optimization according to the sequences of updates recorded in the logs.

3. Incremental Class and Function Definition
   Yew schema objects(C-classes) and functions on C-classes can be added after instances are bound to classes. If the new classes contradict the instances of base C-classes, all the further transactions may be rejected as inconsistent. The contradicting instances of derived C-classes will be automatically fised when the object-binding for the classes is performed.

The current version of the language is quite tentative and will be subjected to many changes in the future.

There are built in classes and functions. For classes, we have 'top', 'bool','number','string', 'sequence','bag', 'set', etc. For functions, we have:

plus	:	*number* × *number*	→	*number*	;	(add numbers)
minus	:	*number* × *number*	→	*number*	;	(subtract a number from a. number)
...						
length	:	*string*	→	*number*	;	(string length)
substring?	:	*string* × *string*	→	*bool*	;	(1st arg. is a substring of 2nd arg.?)
··· . etc.						

The following is the actual session performed on this system. The lines preceded by ";;" are the comments, which were added afterwards. The highlights are in the second half of the session, where the automatic integrity constraints checking, incremental class definition and object-binding are demonstrated.

```
ADL[0]> (lisp (reset-kb!))
                    ;; Clear all instances and initialize transaction management
                    ;; routine.
rest-kb

ADL[0]> (defconcept person (base entity) (isa top)
           ((name string) (address location) (age number) (phone string)
            (occupation string) (salary number))
           (res (and (gt (age self) 0) (lt (age self) 200 )))

ADL[0]> (defconcept location (base entity) (isa top)
           ((state string) (city string) (street string) (number string)
            (apartment string) (apartment-number string))
           (res true))

ADL[0]> (defconcept student (derived entity) (isa person) ()
           (res (equal (occupation self) "student")))

ADL[0]> (defconcept professor (derived entity) (isa person) ()
           (res (equal (occupation self) "professor")))

                    ;; We have defined four new C-classes: person, location, student,
                    ;; and professor.

ADL[0]> (compile)
                    ;; recompile the classes and functions.

;; First, we demonstrate a nested transaction and object sharing.
;;

ADL[0]> (begin-transaction) [1]
                    ;; begin the transaction.
                    ;; The system supports nested transactions.
                    ;; The number in the prompt "ADL[#]>" shows the nesting depth.
ADL[1]> (insert (person (name "John")
                         (address (location (state "CA")
                                            (city "Palo Alto")
                                            (street "Yale")
                                            (number "2260")))
                         (age 20)
                         (salary 40000)))

ADL[1]> (set john (find person (equal (name self) "John")))
                ;; Any instance can be bound to a global variable.
                ;; Note that we don't have to specify the all of the attribute
                ;; values, because an attribute is treated as a partial function.

ADL[1]> (insert (person (name "Mary")
                         (address (location (state "NY")
                                            (city "New York")
                                            (street "West")
                                            (number "47")))
                         (age 18)
                         (salary 50000)))

ADL[1]> (set mary (find person (equal (age self) 18)))

ADL[1]> (end-transaction)
transaction[1] successfully terminated

ADL[0]> (begin-transaction)[2]

ADL[1]> (modify mary age 25)

ADL[1]> (begin-transaction)[3]

ADL[2]> (begin-transaction) [4]

ADL[3]> (modify mary age 21)
                ;; We modified Mary's age in the deepest level of the
                ;; transactions.

ADL[3]> (end-transaction)
transaction[4] successfully terminated

ADL[2]> (output mary)
                ;; We show that Mary's age is actually modified.

[person]:
  salary -> [number]:50000
  age -> [number]:21
  address ->
   [locationl:
     number -> [string] :"47"
     street -> [string]:"West"
     city -> [string] :"New York"
     state -> [string] :"NY"
  name -> [string] :"Mary"
```

```
ADL[2]> (modify mary address (address john))
                ;; Mary's address becomes the same as John's address
                ;; The object is shared.
ADL[2]> (output person)
                ;; Now, both persons have the same address.

Instances[person]:::

  [person]:
    salary -> [number]:50000
    age -> [number]:21
    address ->
      [location]:
        number -> [string]:"2260"
        street -> [string]:"Yale"
        city -> [string]:"Palo Alto"
        state -> [string]:"CA"
    name -> [string]:"Mary"

  [person]:
    salary -> [number]:40000
    age -> [number]:20
    address ->
      [location]:
        number -> [string]:"2260"
        street -> [string]:"Yale"
        city -> [string]:"Palo Alto"
        state -> [string]:"CA"
    name -> [string]:"John"

ADL[2]> (modify (address mary) city "Stanford")
                ;; We change the city of Mary's address to "Stanford".
                ;; Since the location object is shared, this change is
                ;; automatically propagated to John's address.
ADL[2]> (output person)
                ;; The change is actually propagated.
Instances[person]:::

  [person]:
    salary -> [number]:50000
    age -> [number]:21
    address ->
      [location]:
        number -> [string]:"2260"
        street -> [string]:"Yale"
        city -> [string]:"Stanford"
        state -> [string]:"CA"
    name -> [string]:"Mary"

  [person]:
    salary -> [number]:40000
    age -> [number]:20
```

```
      address ->
         [location]:
           number -> [string]:"2260"
           street -> [string]:"Yale"
           city -> [string]:"Stanford"
           state -> [string]:"CA"
      name -> [string]:"John"

 ADL[2]> (modify john age 300)
                 ;; This change contradicts the integrity constraints that
                 ;; a person's age should be greater than 0 and less than 200.
 ADL[2]> (end-transaction)
 transaction[3] aborted
                 ;; The transaction in level 2 is rejected.
                 ;; Since the modification of the addresses of John and Mary
                 ;; are performed in level 2, it is thrown away.
 ADL[1]>
 (modify john age 30)
                 ;; Just one more change in level 1.
 ADL[1]> (end-transaction)
 transaction[2] successfully terminated
                 ;; The only changes performed in level 1 have been accepted.
 ADL[0]> (output person)
                 ;; We show what has been changed.
 Instances[person]I::

    [person]:
      salary -> [number]:50000
      age -> [number]:25
      address ->
         [location]:
           number -> [string]:"47"
           street -> [string]:"West"
           city -> [string]:"New York"
           state -> [string]:"NY"
      name -> [string]:"Mary"

    [person]:
      salary -> [number]:40000
      age -> [number]:30
      address ->
         [location]:
           number -> [string]:"2260"
           street -> [string]:"Yale"
           city -> [string]:"Palo Alto"
           state -> [string]:"CA"
      name -> [string]:"John"
                 ;; Only Mary and John's ages have been changed.

;; Next we demonstrate the automatic object-binding.
;;
ADL[0]> (output student)
```

```
Instances[student]:::
                 ;; No instances are bound to 'student'.

ADL[0]> (begin-transaction)[5]

ADL[1]> (modify john occupation "student")
                 ;; John becomes a 'student'.

ADL[1]> (end-transaction)
transaction[5] successfully terminated

ADL[0]> (output student)
                 ;; Now, John is bound to 'student' as an instance.

Instances[student]:::

  [student]:
    salary -> [number]:40000
    occupation -> [string]:"student"
    age -> [number]:30
    address ->
      [location]:
        number -> [string]:"2260"
        street -> [string]:"Yale"
        city -> [string]:"Palo Alto"
        state -> [string]:"CA"
    name -> [string]:"John"

ADL[0]> (begin-transaction)[6]

ADL[1]> (modify john occupation "professor")
                 ;; John becomes a 'professor'. He is no longer a 'student'.

ADL[1]> (end-transaction)
transaction[6] successfully terminated

ADL[0]> (output student)

Instances[student]:::
                 ;; He is no longer bound to 'student'.

ADL[0]> (output professor)
                 ;; Now he has been moved from 'student' to 'professor'.

Instances[professor]:::

  [professor]:
    salary -> [number]:40000
    occupation -> [string]:"professor"
    age -> [number]:30
    address ->
      [location]:
        number -> [string]:"2260"
```

```
          street -> [string]:"Yale"
          city -> [string] :"Palo Alto"
          state -> [string] :"CA"
      name -> [string] :"John"

;; Next demonstration shows the integrity constraints involving several
;; C-classes.
;;

ADL[0]> (defconcept I-am-the-richest (base entity) (isa top)
          ((name string) (salary number))
          (res (forall ((x person)) (gt (salary self) (salary x)))))

                    ;; First, we define a new C-class, which claims that
                    ;; it is richer than any 'person'.
ADL[0]> (compile)
                    ;; Incrementally compile the schema.

ADL[0]> (begin-transaction)[7]

ADL[1]> (insert (I-am-the-richest (name "tyrant") (salary 10000)))

ADL[1]> (end-transaction)
transaction[7] aborted
                    ;; Since there is already a 'person' whose 'salary' is
                    ;; more than 10000, the transaction is rejected.

ADL[0]> (begin-transaction)[8]

ADL[1]> (insert (I-am-the-richest (name "tyrant") (salary 100000)))

ADL[1]> (end-transaction)
transaction[8] successfully terminated
                    ;; No 'person' earns more than 100000. So, this transaction
                    ;; is accepted.

ADL[0]> (begin-transaction)[9]
                    ;; Now, we try to insert a 'person' whose salary is
                    ;; More than "tyrant."

ADL[1]> (insert (person (name "richman") (age 45) (salary 110000)))

ADL[1]> (end-transaction)
transaction[9] aborted
                    ;; Although, "richman" satisfies the local constraint on
                    ;; the age, this transaction is rejected, because "tyrant"
                    ;; doesn't allow a richer 'person' than him.

;; We can use any first order formula to express the integrity constraints.
;; The following example demonstrates the use of quantified first order formulas.
;; Since the schema objects can be incrementally defined, we can express
;; complicated query by a schema definition.
```

```
ADL[0]> (defconcept oldest-person (derived entity) (isa person) nil
          (res (forall ((x person)) (ge (age self) (age x)))))

ADL[0]> (defconcept the-oldest-person (derived entity) (isa person) nil
          (res (forall ((x person))
                           (if (not (equal self x)) (gt (age self) (age x))))) )
                      ;; Two classes are added. The class 'the-oldest-person'
                      ;; should be a person who is really older than any one else.
ADL[0]> (compile)

ADL[0]> (output oldest-person)
                      ;; Both 'oldest-person' and 'the-oldest-person' has an
                      ;; instance, because there is only one person with the
                      ;; oldest age.
Instances[oldest-person]:::

   [oldest-person]:
     salary -> [number]:40000
     occupation -> [string]:"professor"
     age -> [number]:30
     address ->
        [location]:
          number -> [string]:"2260"
          street -> [string]:"Yale"
          city -> [string]:"Palo Alto"
          state -> [string]:"CA"
     name -> [string]:"John"

ADL[0]> (output the-oldest-person)

Instances[the-oldest-person]:::

   [the-oldest-person]:
     salary -> [number]:40000
     occupation -> [string]:"professor"
     age -> [number]:30
     address ->
        [location]:
          number -> [string]:"2260"
          street -> [string]:"Yale"
          city -> [string]:"Palo Alto"
          state -> [string]:"CA"
     name -> [string]:"John"

ADL[0]> (begin-transaction)[10]
                      ;; Now, we add one more 'person' whose age is the oldest.

ADL[1]> (insert (person (name "Kate") (age 30) (salary 45000)))

ADL[1]> (end-transaction)
transaction[10] successfully terminated
                      ;; Now, there are two persons with the oldest age 30.
```

```
ADL[0]> (output oldest-person)
                        ;; So, 'oldest-person' has two instances.
InstancesColdest-person]:::

  [oldest-person]:
    salary -> [number]:40000
    occupation -> [string]:"professor"
    age -> [number]:30
    address ->
      [location]:
        number -> [string]:"2260"
        street -> [string]:"Yale"
        city -> [string]:"Palo Alto"
        state -> [string]:"CA"
    name -> [string]:"John"

  [oldest-person]:
    salary -> [number]:45000
    age -> [number]:30
    name -> [string]:"Kate"

ADL[0]> (output the-oldest-person)

Instances[the-oldest-person]:::
                        ;; But 'the-oldest-person' has no instances, because
                        ;; there is no person who is strictly older than anyone else.

ADL[0]>
```

References

[AK 89] Abiteboul S. and Kanellakis P.: *"Object Identity as a Query Language Primitive"; In Proc. A CM SIGMOD '89*

[BS 81] Burris S. and Sankappanavar H.: *A Course in Universal Algebra*, Springer-Verlag, 1981

[CC 89] Cacace F, Ceri S, Crespi-Reghizzi S, Tanca L and Zicari R: *" The LOGRES Project: Integrating Object-oriented data modeling with a Rule-based Programming Paradigm"* ; Working paper, June, 1989.

[CW 89 Chen W. and Warren D .: *"C-Logic of Complex Objects"; In Proc. A CM PODS '89.*

[FS 89] Fishman D.H. et al: *"Overview of Iris DBMS"*; HP System Laboratory Technical Report, HPL-SAL-89- 15, 1989.

[GB 85] Goguen J. and Burstall R.: *"Institutions: Abstract Model Theory of Corn-p uter Science."* ; CSLI Technical Report CSLI-85-30, Center for the Study of Language and Information, Standard University, 1985.

[GM 89] Goguen J.A. and Meseguer J.: *"Order-Sorted Algebra I"*; Working paper, July, 1989

[KL 86] Keller A.: *" The Role of Semantics in Translating View Updates"; Computer*, Jan, 1986

[KW 89] Kifer M. and Wu J.: *"A Logic for Object-Oriented Logic Programming"; In Proc. ACM PODS'89.*

[LR 89] Lecluse C. and Richard P.: *"Modeling Complex Structures in Object-Oriented Database"; In Proc. A CM PODS '89.*

[LU 89] Lunt T.F.: *"Multilevel Security for Object-Oriented Database Systems"; IFIP '89.*

[MR. S6 Maier D.: *"A logic for objects"*; TR CS/E-86-012, Oregon Graduate Center Technical Report, 1986

[SP 81] Shipman D.: *" The Functional Data Model and the Data Language DAPLEX"; ACM Trans. on Database System* Vol.6, No.2. 1981

[SS 77] Simth J.M. and Simth C.P.: *"Database Abstraction: aggregation and generalization"* ; *A CM Trans. on Database System*, Vol. 2, No. 2, 1977

[UL 87] Ullman J.D.: *"Database Theory — past and future"; In Proc. Sixth A CM Syrnp. on Principles of Database Systems,* 1987

[UL 88] Ullman .J.D.: *Database and Knowledge-base Systems;* Computer Science Press, 1989

[WH 83] Wiederhold *G.: Database Design;* McGraw-Hill, 1983

[Wh 86] Wiederhold G.: *" Views. Objects and Databases"; Computer*; Dec. 1986

[WT 89] Winston P.H., Horn B.K.: *LISP;* Adisson Wesley, 1989