データベース・システム　78－16
（１９９０．７．２０）

演繹・オブジェクト指向データベース言語 *Juan* の概要

横田 一正
(財) 新世代コンピュータ技術開発機構

演繹データベースにオブジェクト指向概念を組み込んだ演繹・オブジェクト指向データベース (DOOD) のための言語 *Juan* の概要を述べる。論理型言語の枠組で、オブジェクトのデータ構造と手続きが表現されている。オブジェクト識別子は内包的オブジェクトや永続オブジェクトのために拡張項で表現され、その単一化理論としてはレコード代数を用いている。オブジェクトの属性は部分情報の表現に適している。ルールの頭部に現れる属性がメソッドとして用いられ、それも拡張項で表現される。データベースにモジュール概念が導入されており、問合せはモジュール間のメッセージ・パッシングで実行される。本稿では、*Juan* の特徴を中心的に説明する。

Outline of a Deductive and Object-Oriented Language *Juan*

Kazumasa Yokota
Institute for New Generation Computer Technology (ICOT)
21F., Mita-Kokusai Bldg., 1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN
e-mail: kyokota@icot.or.jp

We explain the outline of *Juan*, which is a deductive and object-oriented database language under design, along the line of extensions of deductive databases with object-orientation concepts. In *Juan*, data and procedures are expressed in the framework of logic programming. Object identifiers are in the form of extended terms for intentionally defined or persistent objects, unification of which is formulated in record algebra. Properties of objects are appropriate for partial information. All attributes in a head of a rule is also in the form of extended terms to play a role of methods. Modular concepts are introduced in *Juan* and a query is processed by message passing between modules. In this paper, we describe mainly the features of *Juan*.

1 Overview

There are many approaches,which have been proposed as extensions to relational databases for supporting more application domains, frequently referred as 'new' ones. Among such extensions, deductive and object-oriented databases (DOODs) are expected as a powerful candidate for next generation database systems [14]. Briefly speaking, DOODs are intended to be integrated databases with both advantages of deductive databases (DDBs) and object-oriented databases (OODBs). For such intentions, there are several approaches from some viewpoints: which is regarded more important, DDBs or OODBs?; what are characteristics of OODBs? [2, 4]; what kinds of incomplete (and partial) information should be focused on?

We take an approach for DOODs as extensions of DDBs, by embedding object-orientation concepts into the formal framework of DDBs. Extensions of DDBs are classified from three viewpoints: logical, data modeling (encapsulary), and computational (paradigmatic) extensions [14]. Especially, from viewpoint of the data modeling ones, there are many works in three aspects: introduction of complex data structure, encapsulation of data and procedures, and introduction of object identity.

In this paper, we describe rather *informally* an outline of a DOOD language called *Juan*, which focuses on 'object-oriented (database)' extensions in the above framework: mainly data modeling and computational extensions. Not to mention, we do not exclude other extensions, but we would rather embed also logical ones into next version of *Juan*.

Juan has many distinguished features from other related works such as F-logic [5, 6] and DOT [10], especially in object identity, methods, modules, and inheritance. We discuss how to embed object-orientation features into DDBs in Section 2: object identity, properties, hierarchy, inheritance, and methods, where we do not necessarily intend to cover all its features. After then, we discuss the unification in Section 3 and explain the outline of *Juan* in Section 4, and then describe some points to be further discussed in Section 5.

2 Embedding Object-Orientation Features

2.1 Object Identity

Object identity is a property, which identifies an object from others, and an *object identifier* (*oid*) is representation for object identity. There are some criteria for generating and maintaining *oids*, especially from a viewpoint of DOODs as extensions of DDBs:

(1) Rules define objects intentionally, each of which also should has an *oid*. That is, rules should support a mechanism for dynamically generating the *oid* of such an object. This feature has not been considered sufficiently in other OODBs.

(2) Object sharing needs a 'global' *oid* referred from the related objects, especially in a distributed environment.

(3) A persistent object also needs an *oid*, which should be possible to be recalled when the object is activated in memory again.

(4) An *oid* should be given even when we have only partial information about some object, because we cannot expect an object has a fixed number of attributes and fixed structure as the identification information.

Under such considerations, we define *oids* as extended term representation based on attribute-value pairs, and the corresponding naming mechanism in the form of rules.

For example, consider a graph consisting of *nodes* and *arcs*, where paths are defined in Prolog as follows:

$$path(X,Y) \Leftarrow arc(X,Y).$$
$$path(X,Y) \Leftarrow arc(X,Z), path(Z,Y).$$

What kind of an *oid* should be given to such an intentionally defined *path*? Each *path* can be identified by the corresponding route from the initial node to the terminal one, where an *oid* might contain cyclic routes (infinite structure) in a *path*.

Now such route information can be embedded as a parameter of the predicate *path* in a Hilog style [3]:

$$path([X,Y])/(X,Y) \Leftarrow arc(X,Y).$$
$$path([X|R])/(X,Y) \Leftarrow arc(X,Z), path(R)/(Z,Y).$$

In this case, $path(R)$ before '/' plays a role of the *oid* of a *path* [1] , which is generated by the rules, and $arc(X,Y)$ is an *oid* in itself. As such an *oid* is logically generated,

[1]Note that attributes in *oids* (shortly, *id-attributes*) can be also embedded into a list of properties as follows:

$$path(X,Y,[X,Y]) \Leftarrow arc(X,Y).$$
$$path(X,Y,[X|R]) \Leftarrow arc(X,Z), path(Z,Y,R).$$

differently from pointers or numbers, it can be used over multiple name spaces, especially for sharing objects or activating persistent objects.

Another critical point is to represent partiality of an object. Even if we have not a complete list of properties, we want to identify some object from others. Consider the following example in the form of attribute-value pairs:

$person[name = john]/$[a list of attributes]
$person[name = paul, age = 24]/$[a list of attributes]
$person[name = paul, age = 30]/$[a list of attributes]

where the first *person* is identified only by a *name*, while the second and the third *persons* are discriminated by both *names* and *ages*. When we want to distinguish multiple *johns*, we can add any number of attributes into *oids*.

For coming up to the above four criteria, we represent an *oid* in the form of extended terms, called an *object term*. Assume a set $\mathcal{O}$ of basic objects and a set $\mathcal{V}$ of object variables. An object term is recursively defined on $\mathcal{O} \cup \mathcal{V}$ as follows:

(1) A basic object is an object term,

(2) An object variable is an object term, and

(3) $O[l_1 = o_1, \cdots, l_n = o_n]$ is an object term, where $O, l_1, \cdots, l_n$ are a basic object or an object variable, and $o_1, \cdots, o_n$ are object terms. $l_i = o_i$ is called an *id-attribute*

2.2 Properties

Each object can have any numbers of properties (or attributes), which are different from id-attributes. In *Juan*, a separator '/' is used for separating id-attributes from the succeeding attributes. Such an object is called an attribute term defined as follows:

(1) An object term is an attribute term, and

(2) $O/[F_1\theta_{11}A_1, \cdots, S_1\theta_{21}\{A_{11}, \cdots, A_{1n}\}, \cdots]$ is an attribute term, where O, F_i, S_j are object terms, A_i, A_{2j} are attribute terms $(i, j \geq 0)$, θ_{kl} is called an operator: $\rightarrow$, $\leftarrow$, or $=$.

F_i and S_j play a role of *labels*, F_i is called a function label which takes a *function value*, S_j is called a *set label* which takes a *set value*. Assume that any label is typed such that it takes either a function value or a set value.

For example, consider the following example:

$john/[age = 30, hobby \rightarrow \{music, sport\}, parent \leftarrow \{mary\}]$

where *john*, *age*, 30, *hobby*, *music*, *sport*, *parent*, and *mary* are basic objects, *john* plays a role of an *oid* of the object, '/' is a separator, and there are three attributes in '[' and ']'. 30, *music*, *sport*, *mary* also might play *oids* in other attribute terms, that is, complex objects are represented in a nest of *oids* (although we don't mention it in this paper). A label *age* takes a function value, while *hobby* and *parent* take set values. There are three kinds of operators:

(1) $age = 30$ — *john*'s *age* is exactly 30 years old.

(2) $hobby \rightarrow \{music, sport\}$ — *john*'s *hobby* is subsumed by *music* or *sport*, that is, an operator '$\rightarrow$' represents some possibility of the attribute.

(3) $parent \leftarrow \{mary\}$ — one of *john*'s *parents* is *mary*, that is, an operator '$\leftarrow$' represent partial information of the attribute.

Not to mention, operators '$\rightarrow$' and '$\leftarrow$' may take a function value and '=' may take a set value.

Already mentioned in 2.1, an *oid* has also id-attributes like the following:

$$john[last\_name = lennon]/[\cdots]$$

Such an id-attribute is also treated as one of properties. In another word, a list of id-attributes plays a role of a *primary key* in all properties of an object.

What are differences between id-attributes and attributes? Ullman discussed about value-oriented systems versus object-oriented systems [11]. In his word, id-attributes (or an *oid*) is value-oriented, while attributes succeeding an *oid* are object-oriented, because the former is decided by combination of basic objects and cannot be dissolved, while the latter can be dissolved into conjunction of simpler attribute terms with the same *oid*. In the sense, *Juan* can be said to integrate two approaches.

2.3 Method and the Implemantation

Labels (attribute names) are conventionally formulated as functions from a set of *oids* to a set of attribute values. The formulation can be read from an object-orientation point of view as follows:

- An *oid* is a *receiver object*,
- A label is a *message identifier*,

- Arguments (if there are) of the label are *messages* themselves,
- An attribute value (,which might be a variable) is a *return value*, and
- The *implementation* is written in the body of the rule.

That is, each attribute can be read as a *method.* In F-logic [5], labels are extended to a first order term called an *id-term*, that is, a message with arguments, and in new F-logic [6], the notation is revised like 'natural' methods with any numbers of arguments:

$$person[legal\_names{::}\ P, Y \twoheadrightarrow \{N\}] \longleftarrow person : P[last\_name :: Y \rightarrow N]$$

And the semantics is changed for such variable numbers of variables. However, it is a kind of overloading and the argument positions are left to be fixed because id-terms used as labels are essentially a predicate-based notation.

We extend labels (methods) in the form of extended terms, the syntax of which are same as one of *oids*. Each attribute consists of a message identifier and the message itself. As a message, there can be any numbers of arguments in the form of an attribute-value pair. Even if all messages are not given to an object, the evaluation is proceeded.

A *rule* consists of an attribute term (a *head*) and a set of attribute terms (a *body*), denoted as $Head \Leftarrow B_1, \cdots, B_n$. Any argument used as an attribute in a head can be a method to the object (the *oid*), and the implementation is written in the body. A method is defined as a set of rules with same *oid*, and each rule can has different implementation in each body.

Consider the following example:

$$person[name = john]/[tel[location = X] \rightarrow Y] \Leftarrow \text{implementation part},$$

where assume that object variables X and Y are bound in the implementation part. If a message $tel[location = office]$ or $tel[location = home]$ is sent to an object $person[name = john]$, it is evaluated in the body, and the corresponding phone number is returned as an instance of Y. If a message *tel* without an argument is sent to the object, instances of a pair of X and Y are returned. Further, a phone number might be changed according to time such as daytime and night. Representation in such an extended term is adaptable for such changes.

Even if labels are extended in the form of object terms, the semantics remains to be basically same, that is, the labels are interpreted as functions, the domain of which is a set of *oids* and the range of which is also a set of *oids* or the power set [2].

2.4 Lattice on a Set of Object Terms

In this subsection, we introduce subsumption relation between ground object terms (*oids*), and discuss about the related inheritance of properties between objects (attribute terms).

2.4.1 Ordering between Objects

Assume that a set $\mathcal{O}$ of basic objects has lattice structure (with a join $\wedge$ and a meet $\vee$ operations) based on the *subsumption* ordering $\preceq$ between basic objects. By the definition of an object term in 2.1, a ground object term is also defined as a pair (T, f) of a set T of *trees* constructed on $\mathcal{O}$ [3] and a function f from T to $\mathcal{O}$. Let $\mathcal{T}$ be a set of ground object terms.

Consider the following example:

$$person[name = john, \quad phone = number[office = 11, home = 12]]$$

It is represented as the following pair (T, f):

$$T = \{\top, \top.name, \top.phone, \top.phone.office, \top.phone.home\}$$
$$f = \{(\top, person), (\top.name, john), (\top.phone, number), (\top.phone.office, 11), (\top.phone.home, 12)\}$$

As an example of *path* in 2.1, an object term may have infinite structure.

The subsumption ordering of basic objects is extended to $\mathcal{T}$ as follows:

$$(T_1, f_1) \sqsubseteq (T_2, f_2) \iff T_2 \subseteq T_1 \wedge \forall o \in \mathcal{O}^*.\ f_1(o) \preceq f_2(o).$$

By using the ordering, we can write properties as follows:

$$o/[l \rightarrow o_1, l \leftarrow o_2].$$

where $o_2 \sqsubseteq o_1$. It means a *range* of the attribute value of $o.l$.

We introduce an *equivalence relation* $\cong$ on $\mathcal{T}$. If two object terms o_1 and o_2 have same structure except a

[2]Already mentioned, an attribute value can be separated as combination of the corresponding *oid* (or a set of *oids*) as the attribute value and (a set of) the attribute term(s) with the same *oid*(s).

[3]First, consider a unitary semigroup $\mathcal{O}^*$, which is constructed by $\mathcal{O}$ and a concatenation operator '.'. Then a tree t is defined as a subset of $\mathcal{O}^*$:

$$\forall a, b \in \mathcal{O}^*.\ a.b \in t \supset a \in t.$$

number of cycles or concatenation of part of the cycle, we define $o_1 \cong o_2$ (see the details in [13]). This relation corresponds also to an example of Krebs cycle in [9]. $\sqsubseteq$ is extended by $\cong$ as follows:

$$o_1 \cong o_2 \supset (o_1 \sqsubseteq o_2 \wedge o_2 \sqsubseteq o_1).$$

The relation $\sqsubseteq$ is partial ordering on $\mathcal{T}/\cong$.

2.4.2 Lattice on a Set of Objects

Given two object terms (T_1, f_1) and (T_2, f_2), a new function $f_1 \wedge f_2$ is defined as a minimal set satisfying the following conditions:

(1) $\exists(p, n_1) \in f_1, \exists(p, n_2) \in f_2.\ (p, n_1 \wedge n_2) \in f_1 \wedge f_2$

(2) $\exists(p, n_1) \in f_1, \neg\exists(p, n_2) \in f_2.\ (p, n_1) \in f_1 \wedge f_2$

(3) $\exists(p, n_2) \in f_2. \neg\exists(p, n_1) \in f_1.\ (p, n_2) \in f_1 \wedge f_2$

(4) If $\exists(p_1, \perp) \in f_1 \wedge f_2$, then $\neg\exists(p_2, n) \in f_1 \wedge f_2$, where p_1 is a *prefix* of p_2.

Note that $f_1 \wedge f_2$ is defined even if there are n_1 and n_2 such that $n_1 \wedge n_2 = \perp$ and there is no prefix which takes $\perp$.

Similarly, $f_1 \vee f_2$ is defined as follows:

$$\exists(p, n_1) \in f_1, \exists(p, n_2) \in f_2.\ (p, n_1 \vee n_2) \in f_1 \vee f_2$$

As intersection and union of two trees are also trees, a meet $\sqcup$ and a join $\sqcap$ operations are defined for given two object terms (T_1, f_1) and (T_2, f_2), as follows:

(1) $(T_1, f_1) \sqcup (T_2, f_2) = (T_1 \cap T_2, f_1 \vee f_2)$

(2) $(T_1, f_1) \sqcap (T_2, f_2) = (T_1 \cup T_2, f_1 \wedge f_2)$

Clearly $\mathcal{T}/\cong$ constitutes a *complete lattice.*

2.4.3 Inheritance and Exception

Attributes are inherited along the lattice on a set of object terms. When there are multiple attributes with same *oid* as the result of inheritance, their values are joined or merged according to the operator:

(1) if $o/[l \rightarrow t_1, l \rightarrow t_2]$, then $o/[l \rightarrow t_1 \sqcap t_2]$,

(2) if $o/[l \leftarrow t_1, l \leftarrow t_2]$, then $o/[l \leftarrow t_1 \sqcup t_2]$.

Note that lattice operations is defined between object terms, while they are not defined between attribute terms. Then, as the result, the lower structure of t_1 or t_2 might be lost by this process.

Recall that id-attributes are treated also as part of properties. Assume that some label l appears in an id-attribute of an object o_1 and in an attribute of another object o_2, and $o_1 \sqsubseteq o_2$:

$$o_2/[l\ \theta\ t_2]$$
$$o_1 : \{o_2\}[l = t_1]/[\cdots]$$

where $o_1 : \{o_2\}$ means $o_1 \sqsubseteq o_2$. In this case, o_1 does not inherit the attribute with l from o_2, that is, an id-attribute with l suppresses to inherit attributes with the same label l, that is, it causes an exception.

Consider the following example in *Juan*:

$$\begin{array}{l} bird/[flying = yes] \\ penguin[flying = no] : bird \\ super\_penguin : \{penguin\}. \end{array}$$

An object $penguin[flying = no]$ does not inherit a property $flying = yes$, while $super\_penguin$ inherits $flying = yes$.

As a label has an argument, another expression can be written by using a label with arguments:

$$\begin{array}{l} bird/[flying[who = bird] = yes] \\ penguin : bird/[flying[who = penguin] = no] \\ super\_penguin : penguin/ \\ \qquad [flying[who = super\_penguin] = yes] \end{array}$$

If the inheritance relation is saturated, we have the followings:

$$\begin{array}{l} bird/[flying[who = bird] = yes] \\ penguin : bird/[flying[who = bird] = yes, \\ \qquad\qquad flying[who = penguin] = no] \\ super\_penguin : penguin/[\\ \qquad flying[who = bird] = yes, \\ \qquad flying[who = penguin] = no, \\ \qquad flying[who = super\_penguin] = yes] \end{array}$$

It does not cause any inconsistency, because each label is different from others.

3 Identifiability

An object is discriminated from another by comparing their *oids*, which might have infinite structure. Then, *oids* cannot be compared simply under unique name assumption, because there is a hierarchy between basic objects, and an intentional object has an *oid* with object variables. For such identification, unification between *oids* is introduced.

3.1 Object Identifier as Record Algebra

An object identifier (*oid*) corresponds to a so-called record structure. For a theory for such data structure, especially as a foundation of unification of *partially specified terms* (*PSTs*) in CIL, *record algebra* is proposed [7]. Here we formulate *oids* as record algebra and the unification theory on it. (Refer the details in [7].)

There are two differences between *oids* and *PSTs* in record algebra:

(1) There are basic objects or object variables in all nodes of a tree in an *oid*, while there are tags or parameters only in leaves of a tree in a *PST* (in record algebra).

(2) There is a *glb* between basic objects because they constitute a complete lattice, while there is only identity relation between tags in record algebra.

These differences make a concept of *conflict* for *oids* to be unnecessary.

In terms of record algebra, $\mathcal{O}^*$ is an *access semigroup* which constitutes a tree, and $\mathcal{O}$ is a *merge system*, an element of which is attached to nodes of the tree. Let R be complete $\mathcal{O}^*$-record algebra generated by $\mathcal{O}$ and $\mathcal{V}$ be a set of object variables. Then, $R[\mathcal{V}]$ is record algebra which adds $\mathcal{V}$ to R. Note that, in [7], *PSTs* are restricted in the tagged position, while the formulation of record algebra does not have the restriction essentially, that is, *oids* are formulated as $R[\mathcal{V}]$.

3.2 Unification between Object Identifiers

An *oid* is a subset of $R[\mathcal{V}]$ and a pair $o_1 \bowtie o_2$ of two *oids* is called a *basic constraint*, where $\bowtie$ is mergeable (or informally unifiable). Before the definition of unification, we must define $o' \cdot o$, where $o', o \in R[\mathcal{V}]$. Let l and o' be $l_{11}.\cdots.l_{1n}$ and $(\{l_{11}, \cdots, l_{11}.\cdots.l_{1n}\}, F)$, respectively, where $F = \{(l_{11}, o_{11}), \cdots, (l_{11}.\cdots.l_{1n}, o_{1n})\}$ (l is called a *branch* of F). And let

$$o = (\{l_{21}, \cdots, l_{2m}\}, \{(l'_{21}, o'_{21}), \cdots, (l'_{2k}, o'_{2k})\}).$$

Then $o' \cdot o$ is defined as

$$(\{l.l_{21}, \cdots, l.l_{2m}\}, F \cup \{(l.l'_{21}, o'_{21}), \cdots, (l.l'_{2k}, o'_{2k})\}).$$

Clearly $o' \cdot o$ is also an element of $R[\mathcal{V}]$. Note that the definition is different from [7], because all nodes take basic objects or object variables. In record algebra, there are seven *constraint axioms* in order to solve a set of constraints:

reflective law	$o \bowtie o$
symmetric law	$o_1 \bowtie o_2 \Rightarrow o_2 \bowtie o_1$
restricted transitive law	$x \bowtie o_1, x \bowtie o_2 \Rightarrow o_1 \bowtie o_2$, where x is an object variable.
base law	$o'_1 \cdot o_1 \bowtie o'_2 \cdot o_2$, where either branches of o'_1 or o'_2 is not a prefix of another.
merge law (1)	$(o_1, o_2) \bowtie o_3 \Rightarrow o_1 \bowtie o_3$
merge law (2)	$(o_1, o_2) \bowtie o_3 \Rightarrow o_1 \bowtie o_2$
cancellation law	$o' \cdot o_1 \bowtie o' \cdot o_2 \Rightarrow o_1 \bowtie o_2$

where (o_1, o_2) is a pair of two elements in a set and $L \Rightarrow R$ means that if L exists, then R also must exist. In these axioms, we do not need the *base law*, because an *oid* has basic objects or object variables in all nodes.

It is easy to understand that these axioms correspond to unification process. For example, the *cancellation axiom* generates new basic constraints between subterms.

Unification algorithm is simply given as follows [7]:

(1) An input is a set $C = \{c_1, \cdots, c_n\}$ of basic constraints.

(2) Saturate C by applying the above axioms. This step stops at finite steps, and results in $\overline{C}$.

(3) $\overline{C}$ is the required output.

In a case of *oids*, this algorithm always succeeds where $\overline{C}$ contains a set of substitutions. However, further computation is needed:

(1) As a *glb* of two basic objects is not computed during the process, we must compute $\overline{C}$ by lattice operations of $\mathcal{O}$.

(2) As $\perp$ is normally computed during the process, we should cut down some cases.

Note that unification between *oids* can be formulated in another way. That is, as an *oid* can be translated into a ψ-term [1], we can consider the unification as the join operation (and the generalization as the meet operation).

4 Outline of Juan

In the previous sections, we explain various features of *Juan*. In this section, we sketch the syntax and the semantics, the details of which will be explained in [13].

4.1 Syntax

There are four kinds of symbols: a set $\mathcal{O}$ of basic objects, a set $\mathcal{V}$ of object variables, a set $\mathcal{W}$ of world identifiers, and a set $\mathcal{R}$ of rule identifiers. We show the syntax briefly in the BNF style in Table 1..

In the above syntax, we have not explained some features yet:

- Introducing a *dotted object* $o_1.o_2.\cdots.o_n$, which is useful for indirect representation of an object. For example, $john/[affiliation \rightarrow paul.affiliation]$.
- Introducing a *world* concept and modularizing a database. See the details in [12, 8].

We have some additional restrictions to the syntax:

- When basic objects and object terms are used as labels, they are typed for deciding to take a function value or a set value.
- For avoiding A-unification, object variables in a dotted object are restricted in unification.
- Object variables cannot appear in a set value, for avoiding ACI-unification.

The overall structure of *Juan*, defined above, consists of six-level entities as Table 2. And the ordering between object terms and between words are defined as $o : \{o_1, \cdots, o_n\}$ and $w : \{w_1, \cdots, w_m\}$ respectively.

4.2 Semantics

There can be some approaches for the semantics of *Juan*. If object terms and attribute terms can be interpreted as a set of constraints, just like as a set of semantic expressions in [10], we can consider *Juan* as a constraint logic programming language CLP(X). However, in the case, there is a problem on treating rules:

- If a rule is written as a triple of a head, a body, and a set of constraints, where a head and an element of a body are based on predicate-based notation, it is easy to consider *Juan* as one of CLP(X). However, it might be ambiguous why extended term representation is introduced.
- A rule itself can be considered as a Boolean constraint. However it is questionable whether the corresponding constraint solver works efficiently or not, because implication in a rule might disappear.

[4]'α list' means 'α {","α}'.

So, we consider the semantic domain U as a set of ground object terms, that is, a mapping $\mathcal{M}$ interprets $o \in \mathcal{T}$ as an element $o \in U$ [5]. In the sense, we abuse $\mathcal{T}$ instead of U and also other symbols.

In this subsection, we sketch the semantics of object terms and attribute terms. As for a modularized database, refer [8]. (The details of the semantics of the whole language will be explained in [13].)

4.2.1 Object Terms

The interpretation is different depending on whether an object term is used as an *oid* or a label. Because of the *macro* level domain, we consider only a case of a label. A label is interpreted as a function from $\mathcal{T}$ to $\mathcal{T}$ or $2^{\mathcal{T}}$, according to the type of the label.

For example, interpretations of $person[name = john]$ and $person/[name = john]$ are different:

$$\begin{aligned}
&\mathcal{M}[\![person[name = john]]\!] \\
&\qquad = \mathcal{M}[\![person]\!][\mathcal{M}[\![name]\!] = \mathcal{M}[\![john]\!]] \in U, \\
&\mathcal{M}[\![person/[name = john]]\!] \\
&\qquad = \mathcal{M}[\![name]\!](\mathcal{M}[\![person]\!]) = \mathcal{M}[\![john]\!] \in U,
\end{aligned}$$

where in the latter $name$ is interpreted as a function from U to U. That is, interpretation of an object term is changed according to what role it plays.

Consider another example:

$$\begin{aligned}
person[name = john]/[&tel[location = home] = 11, \\
&tel[location = office] = 12]
\end{aligned}$$

The interpretation constitutes a kind of *hyper-graph*:

$$\begin{tikzcd}
& \mathcal{M}[\![person[name = john]]\!] \arrow[dl, "{\mathcal{M}[\![tel[location = home]]\!]}"'] \arrow[dr, "{\mathcal{M}[\![tel[location = office]]\!]}"] & \\
\mathcal{M}[\![11]\!] & & \mathcal{M}[\![12]\!]
\end{tikzcd}$$

where

$$\begin{aligned}
&\mathcal{M}[\![tel[location = home]]\!] \\
&\qquad (\mathcal{M}[\![person[name = john]]\!]) = \mathcal{M}[\![11]\!] \text{ and} \\
&\mathcal{M}[\![tel[location = office]]\!] \\
&\qquad (\mathcal{M}[\![person[name = john]]\!]) = \mathcal{M}[\![12]\!].
\end{aligned}$$

[5]The interpretation $\mathcal{M}$ of an *oid* $(T, f) = (\{l_1.\cdots.l_n, \cdots\}, \{(l_1.\cdots.l_n, o), \cdots\})$ is given as follows:

$$\begin{aligned}
\mathcal{M}[\![(T, f)]\!] &= (\mathcal{M}[\![T]\!], [\![f]\!]) \\
&= (\{\mathcal{M}[\![l_1]\!].\cdots.\mathcal{M}[\![l_n]\!], \cdots\}, \{(\mathcal{M}[\![l_1]\!].\cdots.\mathcal{M}[\![l_n]\!], \mathcal{M}[\![o]\!]), \cdots\})
\end{aligned}$$

That is, it is also a tree in the domain U.

Table 1: Syntax of the Language

<object term>	::=	<basic object>[“[”<id-attribute> list [4]“]”]
		| <object variable>
		| <object term>“.”<object term>
<id-attribute>	::=	<basic object> “=” <object term>
<attribute term>	::=	<object term> [“:”“{”<object term> list“}”] [“/”“[”<attribute> list“]”]
<attribute>	::=	<object term><operator><attribute term>
		| <object term><operator> “{”<attribute term> list“}”
<operator>	::=	“→” | “←” | “=”
<database>	::=	“{“<world rule> list“}”
<world rule>	::=	[<world label>“⟹”] <rule>
		| <world relation>
<world label>	::=	“(”<world identifier>[“,”<rule identifier>]“)”
	::=	| <world label>“/”<world label>
<rule>	::=	<attribute term> [“⇐”<attribute term> list]
<world relation>	::=	<world identifier>“:” “{”<world identifier> list“}”

Table 2: Language Structure

Entities		Definitions
world	=	module in a modularized database
database	=	set of rules
rule	=	(attribute term, {attribute term,···,attribute term})
attribute term	=	(object term, {attribute,···,attribute})
object term	=	(basic object, id-attribute) | (object variable, id-attribute)
basic object, object variable		

4.2.2 Attribute Terms

First, complex attribute term is dissolved into conjunction of simple attribute terms. Let η be a function such that $\eta(o : \{\cdots\}/[\cdots]) = o$. Then

$$\begin{aligned}\mathcal{M}[\![o_1 : \{o_{11},\cdots,o_{1n}\}/[f_1\theta_1 v_1,\cdots,s_1\theta_2\{v_{21},\cdots,v_{2m}\}]]\!] = \\ \mathcal{M}[\![o_1 : \{o_{11}\}]\!] \wedge \cdots \wedge \mathcal{M}[\![o_1 : \{o_{1n}\}]\!] \wedge \\ \mathcal{M}[\![o_1/[f_1\theta_1\eta(v_1)]]\!] \wedge \mathcal{M}[\![v_1]\!] \wedge \cdots \wedge \\ \mathcal{M}[\![o_1/[s_1\theta_2\{\eta(v_{21}),\cdots,\eta(v_{2m})\}]]\!] \\ \mathcal{M}[\![v_{21}]\!] \wedge \cdots \wedge \mathcal{M}[\![v_{2m}]\!] \wedge \cdots.\end{aligned}$$

Next, ordering and operators are interpreted as follows:

(1) $\mathcal{M}[\![o_1 : \{o_2\}]\!] = \mathcal{M}[\![o_1]\!] \sqsubseteq_U \mathcal{M}[\![o_2]\!]$

(2) $\mathcal{M}[\![o_1/[f_1 \rightarrow o_2]]\!] = \mathcal{M}[\![f_1]\!](\mathcal{M}[\![o_1]\!]) \sqsubseteq_U \mathcal{M}[\![o_2]\!]$

(3) $\mathcal{M}[\![o_1/[s_1 \rightarrow \{v_{21},\cdots,v_{2m}\}]]\!]$
$= \forall e_1 \in \mathcal{M}[\![s_1]\!](\mathcal{M}[\![o_1]\!]), \exists e_2 \in \mathcal{M}[\![v_{2i}]\!].\ e_1 \sqsubseteq_U e_2.$
That is, in the case of a set value, the operator is interpreted as *Hoare's ordering*.

(4) '←' is interpreted in the opposite direction and '=' is interpreted as '→'∧'←'

As an attribute term is dissolved into a set of non-nested attribute terms, each of which has only one attribute, we consider only the simple case.

5 Concluding Remarks

We describe the outline of a DOOD language *Juan* under design. *Juan* has capability of various features of object-orientation concepts, which are embedded into DDBs. The key point in such extensions is extended term representation, and we use it to represent not only data structure but also object identity and methods. Another important aspect of an object orientation paradigm is the computational model, which is mapped into query processing in a modularized database.

In this paper, we have not explained the following points:

- Discrimination of function values and set values, where the question is syntactical differences [6] or explicit typing.
- Treatment of dotted objects, which would closely relate to the depths of the language: lattice construction, unification, semantics, and termination of query processing.
- Dynamically generated ordering or inference about a hierarchy itself, where reconstruction of a lattice of object terms might be fallen into inconsistency.
- Semantics of rules, databases, and worlds, which are under consideration, as well as procedural and fixpoint semantics.

As there have been very few works on DOOD languages and the systems, there remain many problems to be discussed. In the tentative syntax of *Juan*, we are making some experiments on molecular biological databases such as [9] and case bases for legal reasoning, through which we are making requirements clear.

Acknowledgments

The author would like to thank Hideki Yasukawa, Yukihiro Morita, members of *Quixote* meeting, and members of ETR-SWG for valuable and stimulating comments and suggestions.

References

[1] H. Aït-Kaci, "An Algebraic Semantics Approach to the Effective Resolution of Type Equations", *TCS*, vol.45, pp.293-351, 1986.

[2] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, "The Object-Oriented Database System Manifesto", *DOOD89*.

[3] W. Chen, M. Kifer, and D.S. Warren, "Hilog as a Platform for Database Languages (or why predicate calculus is not enough)", *DBPL'89*.

[4] The Committee for Rational Thinking, "The Object-Oriented Counter Manifesto", *a manuscript distributed at SIGMOD'90*.

[5] M. Kifer and G. Lausen, "F-Logic: A Higher-Order Languages for Reasoning about Objects, Inheritance, and Scheme", *SIGMOD'89*.

[6] M. Kifer, G. Lausen, and J. Wu, "Logical Foundations for Object-Oriented and Frame-Based Languages", *draft*, 1990.

[7] K. Mukai, "Merge Structure with Semi-Group Operation and its Unification Theory", *Computer Software*, vol.7, no.2, 1990 (in Japanese).

[8] C. Takahashi, "A Deductive Database with Hierarchical Structure", *Proc. of SIGDBS*, IPSJ, Sapporo, July, 1990 (in Japanese).

[9] H. Tanaka, "Metabolic Reaction Database", *Proc. of SIGDBS*, IPSJ, Sapporo, July, 1990 (in Japanese).

[10] M. Tsukamoto, S. Nishio, and T. Hasegawa, "DOT: Term Representation for Logic Databases with Object-Oriented Concepts", *Advanced Database System Symposium*, Kyoto, Dec. 7-8, 1990 (in Japanese).

[11] J.D. Ullman, "Database Theory: Past and Future", *PODS'87*.

[12] K. Yokota, "Deductive Databases with Hierarchical Structure", *ICOT Internal memo.*, pre-session of DOO-WG, June 29, 1989. (in Japanese)

[13] K. Yokota, "A Deductive and Object-Oriented Database Language *Juan*", *in preparation*, 1990.

[14] K. Yokota and S. Nishio, "Towards Integration of Deductive Databases and Object-Oriented Databases: A Limited Survey", *Advanced Database System Symposium*, Kyoto, Dec. 7-8, 1989.