경영정보학연구
제 4 권 2호
1994년 12월

O^2LDM : 객체지향 논리 데이타모형을 위한 언어

정 철 용[1)]

O^2LDM : A Language for Object-Oriented Logic Data Modeling

In this paper we describe a new data modeling language we call O^2LDM. O^2LDM incorporates features from object-oriented and logic approaches. In O^2LDM there is a rich collection of objects organized in a type hierarchy. It is possible to compose queries that involve field selection, function application and other constructs which transcend the usual, strictly syntactic, matching of PROLOG. We give the features of O^2LDM and motivate its utility for conceptual modeling. We have a prototype implementation for the language, which we have written in ML. In this paper we describe an executable semantics of the deductive process used in the language. We work some examples to illustrate the expressive power of the language, and compare O^2LDM to PROLOG.

1) 상명여자대학교 경영학과 조교수

I. Introduction

In this paper we describe a new data modeling language, O^2LDM, which integrates features from two major data modeling paradigms, object-oriented and relational (logic). The language is designed to serve as a basis for developing knowledge-based systems. Its most novel aspect is the enhanced unification algorithm for making deduction in the presence of structured objects.

Several researchers have argued for incorporating many paradigms within the same environment to provide all the constructs necessary for building knowledge-based systems [Borgida 1985, Fikes 1985]. Numerous multiparadigm environments for developing knowledge-based systems have been built; the most notable ones being LOOPS [Bobrow 1983, Bobrow 1985] and KEE [Fikes 1985]. However, these multiparadigm programming environments are usually integrated by embedding one or more paradigms on top of another. This makes it difficult to understand all the interactions between features. We have tried to identify the language requirements of conceptual modeling or knowledge-based programming and build them into the language from the outset. In this respect O^2LDM is like the language FOOPlog[Goguen 1987]. We have observed that representing arbitrary relations among objects and reasoning based on these arbitrary relations are often difficult to express. Furthermore, type systems are often not capable of detecting trivial errors. With type information the system can detect certain kinds of errors that conflict in an obvious manner with the conceptual model. Typed languages are sure to play an important role in conceptual modeling and high-level database languages. A useful survey can be found in [Atkinson 1987].

Our goal was to study the semantics of combining the paradigms in a strongly typed language. In particular we wanted O^2LDM :

- to build inheritance into the logical inference process;
- to provide primitives for modeling domain objects;
- to allow for function definitions and applications;
- to allow arbitrary objects to stand in relations; and
- to provide compile-time type checking.

O^2LDM was greatly influenced by the language Galileo [Albano 1985] and ML [Milner 1990]. Another language with similar goals is Machiavelli [Ohori 1989]. Like O^2LDM, Machiavelli is designed around a collection of statically typed objects which make it well suited for conceptual modeling, and also influenced by ML. One important contribution of Machiavelli is the automatic type reconstruction. Machiavelli is also able to express queries involving complex objects with naturalness.

The BiggerTalk[Gullichsen 1985] system is also related. Here the goal is to build an object-oriented interface in PROLOG based on message passing. The programmer can dynamically restructure the classification of objects. This paper is organized as follows. The next section introduces the language syntax and presents a simple example to give the flavor of the language. Also O^2LDM is compared with PROLOG in modeling domain knowledge. The third section gives an executable semantics of the deductive process used in O^2LDM.

Ⅱ. The O^2LDM Language

Programming in O^2LDM can be divided into three fundamentally distinct phases. The first phase involves the specification of the domain. The domain is specified by a series of type and relation declarations. Type declarations descri0be the various entity types in the domain. Relation declarations specify associations between entity types in the domain.

In the second phase the individual objects, facts and rules for a particular instance of the domain are entered. Domain objects are represented in the systems by creating instances of entity types. A domain object, once created, can be bound to an identifier; any subsequent reference to the identifier in the system is a reference to the entity that the identifier denotes.

Facts state associations among objects in the domain. Rules are used to express facts that depend on other facts. All the domain objects, facts, and rules entered in this phase must be type consistent with the domain specifications.

Through type checking the language ensures this type consistency. In the third phase the user enters into an interactive dialogue with the system. The user enters either an expression or query into the system and the system responds with one or more solutions. The system's response to an expression is its value. If a query is entered, the system responds by finding the solutions that satisfy the query. Again, the type consistency of the expressions and queries entered is checked against the domain specifications.

In this section of the paper, we present an overview of the constructs in O^2LDM. We discuss types, multiple inheritance, expressions, predicates, facts, rules, and queries. The complete syntax of the language is summarized in Figures 1 and 2. Figure 1 is the syntax for type declarations, and the syntax for expressions. Figure 2 illustrates the syntax for predicate signatures, facts, rules, and queries.

The following notational conventions are used in the figures :

- Keywords are in bold.
- For any syntax class S,
  we define S_1st::=S_1, ······ , S_n.
- For any syntax class T,
  we define T_seq::=T_ ; ······ ; T_n.

1. Expressions and their types

The language has the usual sorts of constructs: records, variant records, functions, etc. The language is essentially identical to the language studied by Cardelli [cardelli 1988] with the addition of relations.

Records are unordered, labeled sets of values: [a:=3; b:=true] has type [a: int; b:bool], where a and b are labels. In general a record [1_1:=e_1; ······ ; 1_n: =e_n] has a type [1_1: τ_1; ······ ; 1_n:τ_n], where 1_1, ······ , 1_n are labels, e_1, ······ , e_n are expressions, τ_1, ······ , τ_n are types, and where the expressions e_1, ······ , e_n are of types τ_1, ······, τ_n, respectively.

Labels are a separate domain; they are not identifiers or strings, and they cannot be computed by an expression in the language. While records are labeled cartesian products, a variant is a labeled disjoint sum. A variant type looks syntactically like a record type; it is an unordered set of label-type pairs, enclosed in curly braces instead of brackets. An element of a variant type is a labeled explression, where the label is one of the labels in the variant type, and the expression has a type matching the type associated with that label.

The case statement is used to manipulate variant records in a type-safe manner. For example, an object of type {a:int; b:bool} is either an integer labeled a or a boolean labeled b. Thus the two objects {a:=5} and {b:=false} are objects of type {a:int; b:bool}.

A function can be defined in the language using the lambda expression of the form: fun(x:τ).e, where x is the parame

```
type-binding::=
        type ident ⇔ type

type::=
        ident   |
        bool    |
        int     |
        string  |
        [field-type-seq] |
        {field-type-seq} |
        type → type

field-type::=
        label:type

ident-binding::=
        val ident ⇔ expr

expr::=
        boolean      |
        integer      |
        string-constant |
        ident        |
        object       |
        logic-var    |
        expr.label   |
        case expr of discrim-seq endcase |
        expr(expr) |
        fun(ident:type).expr |
        expr:type

discrim::=
        label::ident ⇒ expr

object::=
        [field_seq] |
        {field}

field::=
        label:=expr
```

Figure 1 : Type declarations and expressions

ter of the function whose type is τ, and where e is an expression — given in terms of the parameter — that is used to compute the value of the function. Lambda expressions in the language evaluate to functions. The type of the function fun(x:τ).e is $\tau \rightarrow \sigma$, where τ is the type of the argument, and σ the type of the result of function evaluation. For the sake of simplicity we allow for only unary functions in the language, i.e., functions with only one argument.

The language does not allow recursive type or function definitions. Since functions in the language are first–class values, they may be passed as arguments to other functions or be the result of function evaluations. A function is treated like any other object in the system. Hence, records in the language can have functional components.

2. Multiple Inheritance

There is an implicit inheritance relation between types that is based on the structure of the types. This is in contrast to some object–oriented languages, such as Smalltalk[Goldberg 1983] where classes are explicitly named and where the inheritance relation between classes is explicitly declared by the user.

We say that a type is a subtype of or is included in another type τ when all the objects denoted by type are also objects denoted by type τ. Subtyping on record types corresponds roughly to the concept of inheritance in object–oriented languages. This correspondence is due to Cardelli [Cardelli 1988], and is now widespread in statically typed object–oriented languages. This notion of subtyping based on records permits a type to have supertypes which themselves are not related. Such objects inherit attributes from multiple sources.

Type declarations in the language introduce names for type expressions. Names for type expressions serve as abbreviations; they do not create new types. We use structural equivalence on types: two types are equivalent in the language only when they have the same structure.

3. Logic programming

In the language, facts and rules are used to assert the relationships between objects. Given facts and rules, queries in the language are used to find unknown objects that satisfy one or more relations.

(1) Literals

A literal in the language is an n-ary predicate of the following form:

p(e_1, …… , e_n)

where p is a predicate symbol, and the argument e_i's are valid expressions in the language. Although literals in the language appear to be similar to their counterparts in PROLOG, there are important differences in their usage and interpretation. In O^2LDM, field selection, function application and other constructs can be used in a literal, even in conjunction with variables.

Another difference between PROLOG and O^2LDM literals, is that in O^2LDM the arguments of a literal are restricted to be expressions from specific domains, i.e., the arguments of a literal are typed. The type of the arguments of a literal depends upon the signature of the predicate. This is different from PROLOG, where there are no restrictions on the arguments of a literal.

Every literal in the program can be statically checked to verify that the type of the arguments do not conflict with the signature specification of the predicate. A disadvantage of restricting the arguments of a literal to specific domains is that it is no longer possible to consider the entire untyped collection of objects as a whole. However, this appears to be of little use, especially if the type structure is expressive and convenient.

(2) Predicate Signatures

In the language the signature or type of every predicate must be specified prior to its use in a literal. For example, the following signature specification in the language:

Signature p(α, [b:β])

states that the signature of the predicate p is a binary relation (or predicate) between expressions of type and the record type [b:β]. Every occurrence of the predicate symbol, p, in a relation must have exactly two arguments and their types must be , and [b:β] respectively.

Let o_1 and o_2 be objects of type α, o_3 and o_4 objects of type β, and o_5 and o_6 objects of type δ. The expression [b := o_3] is a record of type [b:β]. The following is a well-formed literal in the language:

p(o_1, [b:=o_3])

Let the identifier m be bound to record [b := o_4] in the environment, and identifier n to the record [d := o_5]. If f is a function whose type is [b:β] $\rightarrow$ β, and the predicate q is a binary relation of types β and [d:δ], here are some additional examples of well–formed literals:

q(m.b, n)
q(f(m), [d:=o_6])

The argument m.b in the first literal is field selection, and f(m) in the second literal is function application. The expression m.b, n, and f(m) in the above literals are not in normal form; they are shown in the form a user may enter them. The system, however, replaces the above expressions by their normal forms, when it stores the literals in the knowledge base. Let r be a predicate whose signature is a binary relation of type and . Clearly any object of type can participate in a relation that involves the predicate r. Since type is a subtype of type , objects of type can also participate in relations that involve the predicate r. For example, if s is a binary predicate whose signature is:

signature s([a:α], β)

and *g* a function whose type is [b:β] $\rightarrow$ β, the following is a well–formed literal:

s([a:=o_1; b:=o_3], *g*([a:=o_2; b:=o_3]))

```
predicate–signature::=
        signature ident (type_lst)
fact–defn::=
        fact litera
        let logic–var–decl_seq in fact literal
rule–defn::=
        let logic–var–decl_seq in rule literal ⇔ prop_lst
query::=
        let logic–var–decl_seq in list expr such that prop_lst
logic–var–decl::=
        ogic–var : type
prop::=
        literal
        condition
literal::=
        ident(expr_lst)
condition::=
        expr=expr
        expr!!=expr
```

Figure 2 : Signature, Facts, Rules, and Queries

In the above literal, the function g can be applied to the argument [a: =o_1; b: = o_3], because the type of the argument is a subtype of the domain type of the function. The above literal is well-formed because, the type of the first argument is a subtype of [a:α], and the result of function, g, is of type β.

It is important to reiterate that anywhere an expression of type is allowed it can be replaced by an expression of type τ, if τ is a subtype of σ.

(3) Clauses

A clause in the language has the form:

A ⇔ B_1, B_2, …… , B_n

This means that A is provable if all the B_i are provable. The literal A is the head of the clause, and the B_1, …… , B_n is the body.

(4) Facts and Rules

We saw in the preceding section that the signature of every relation (predicate) is to be specified prior to its use in a literal. In the language there are two types of relations–primitive and defined relations. Primitive relations are relations that denote associations among entity types in the domain. These associations are independent of other relations in the domain. Primitive relations are usually defined during the first phase of programming in the language. We declare a primitive relation by specifying its signature.

Once primitive relations have been declared, facts based on these primitive relations can be asserted. Facts state associations among individual entities, not among the types of the entities. Consider the following predicate signatures:

signature q1([b:β], [d:δ])
signature q2(α, [b:β])

Let o_1 and o_2 be objects of type α, o_3 and o_4 objects of type β, and o_5 an object of type δ. Suppose we want to specify that the two objects [b: =o_3] and [d: =o_5] stand in the relation q1. We do this in the language by asserting the following fact:

fact q1([b: =o_3], [d: =o_5])

The arguments of the above fact are consistent with the signature of the predicate q1.

Suppose we want to specify that all objects of type [b:β] in the domain are related to the object [d: =o_5] by the relationship q1. We do this by asserting the following fact:

let U:[b:β] in fact q1(U, [d := o_5])

The logic variable U stands for an object of type [b:β]. Every logic variable in the language is typed, and its syntactic scope is limited to the fact or rule that follows the declaration. Logic variables in the language are denoted by upper case letters.

Suppose m is an identifier that is bound to the record [a:=o_1; d:=o_5] in the environment. The following fact asserts that the field labeled a in the record, identified by m, is related to the object [b:=o_4] by the relationship q2:

fact q2(m.a, [b := o_4])

In the above fact the expression m.a is not in normal form; it is shown in the form the user enters. The system, however, stores the fact with the expression m.a replaced by its normal form:

fact q2(o_1, [b:=o_4])

Defined relations are relations that denote associations among entities also. But unlike primitive relations, these associations are dependent on other relations. In other words, the relation is defined in terms of conditions or constraints that, when met, implies that the relationship holds. Defined relations are more commonly referred to as rules.

To state rules in the language, we need logic variables. A logic variable stands for the same object wherever it occurs in the rule. Consider the following rule in the language:

let P:[a:α], Q:[b:β] in
rule q3(P, Q.b) $\Leftrightarrow$ q1(Q, [d:=o_5]),
q2(P.a, Q)

where q3 is the defined relation–the head of the rule.

The relations q1 and q2 are the conditions that need to be satisfied for q3 to hold – the body of the rule. It is important to note, that it was not necessary to specify the signature of the predicate q3 prior to the rule definition. Since the signature of q3 is dependent on the types of the expressions P and Q.b, it can be easily determined. The signature of q3 is the binary relation [a:α] and β. In fact, the declarations of type may not be necessary at all in the language.

We can define any number of rules for the same defined relation, but their signatures must all be consistent. When we say consistent, we mean that once the signature of a defined relation has been fixed, the signature of the defined relation in all subsequent rules must either match exactly with the original signature, or be a subtype of it. A n-ary signature is a subtype of another n-ary signature if all corresponding components are subtypes.

For example, the following is a rule for the relation q_4:

let X:[a:α; e:σ], Z:[d:δ] in
rule q_4(X, o_4) $\Leftrightarrow$ q1([b:=o_3], Z),
q2(X.a, [b:=o_4])

The rule reads: for all objects of type [a:α; e:σ] if there exists an object of type [d:δ] such that q1([b:=o_3], Z) and q2 (X.a,[b:=o_4]), then q-4(X, o_4).

A logic variable that appears in the head of a rule is universally quantified over objects in its domain. Logic variables that occur in the body, but not in the head, act as if existentially quantified over their respective domains.

(5) Queries

Given facts and rules in the language, queries in the language are used to find unknown objects that satisfy one or more relations. Logic variables are once again used for this purpose.

By invoking a query that contains logic variables one can solve for the variables. The result of a query is the list of substitutions for the variables that satisfy the query. The following is a query in the language:

let M:[a:α], N:β in
list M such that q3(M, N)

The above query finds the substitutions for the logic variable M that satisfy the relationship q3(M,N). The result of the query is a list of substitutions for the

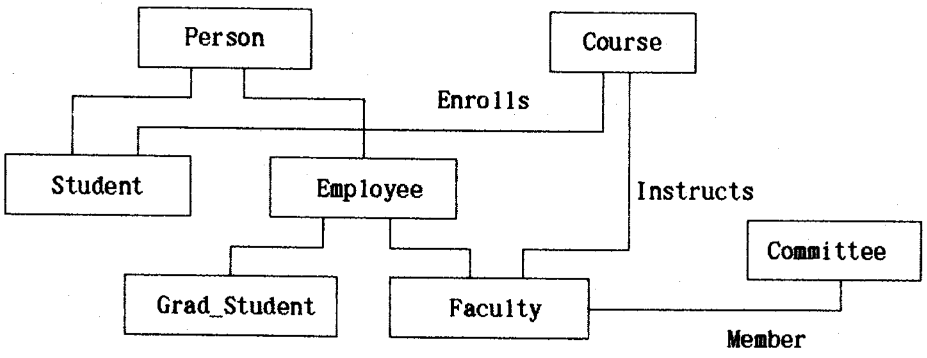


Figure 3 : Conceptual model of university life

logic variable M. The literal q3(M,N) is known as the goal of the query.

A query can have more than one literal. If it has more than one literal, then all the literals have to be satisfied for the query to be satisfied. The system uses the known clauses—facts and rules—to satisfy the goals of a query.

4. An Example

In order to give a flavor of O^2LDM we present an example taken from academic life. Figure 3 illustrates the conceptual model of a facet of university life. The rectangles in the figure represent entity types, the solid lines denote inheritance between entity types, and dotted lines represent relationships between entity types.

In the first phase of programming in the language we specify the conceptual model. We begin by describing the various entity types in the domain.

```
type person ⇔ [name: string; id: int]
type student ⇔ person and [gpa: int]
type employee ⇔ person and [dept:
                      string]
type faculty ⇔ employee and [rank:
                      string]
type grad ⇔ [name: string; id: int;
                      gpa: int; dept: string;
                      adviser: faculty]
type course ⇔ [dept: string; number:
                      int]
type committee ⇔ [name: string; dept:
                      string; room: int]
```

The entity type faculty is almost like the entity type employee. It inherits all the attributes of type employee and has additional attribute rank. Another important aspect of the model are the relationships among entities of the previously defined types. We specify the signature of the primitive relations in the domain as follows:

```
signature instructs(faculty, course)
signature enrolls(student, course)
signature member(faculty, committee)
```

The signature of a relation is a statement of the type of its participants. The order of the participants is important. For example, in the enrolls relation, the first argument is of type student, and the second argument of type course. Naturally, it makes no sense for a committee to be enrolled in a course.

In the second phase of programming the individual objects, facts and rules for a particular instance of a domain are entered. We make the following definition of individual objects for the model of university life.

```
val smith ⇔ [name := "smith"; id :=
        1361; dept := "chemistry"]
val john ⇔ [name := "john"; id :=
        8644; gpa := 5]
val peter ⇔ [name := "peter"; id :=
        2865; dept := "mis"; rank :=
        "prof"]
val nancy ⇔ [name := "nancy"; id :=
        4354; dept := "cs"; rank :=
        "asst"]
val tim ⇔ [name := "tim"; id :=
        2454; gpa := 6; dept := "mis";
        adviser := peter]

val mis600 ⇔ [dept := "mis"; numbe
        r := 600]
val cs565 ⇔ [dept := "cs"; number :
        = 565]
val cs502 ⇔ [dept := "cs"; number :
        = 502]
```

All the objects in this example happen to be records. None of them have functional components. However, O^{2}LDM does allow records to possess functional components.

Relationships between objects are specified by asserting facts. For our example, we assert the following facts:

```
fact instructs (peter, mis600)
fact instructs (nancy, cs565)
fact instructs (nancy, cs502)
fact enrolls (john, cs565)
fact enrolls (john, cs502)
fact enrolls (tim, mis600)
fact enrolls (tim, cs565)
```

A rule defines a new relation in terms of existing relations. For instance, to express the following statement: "A faculty teaches a student if the student is enrolled in a course that the faculty member instructs." we define the rule:

```
let F: faculty; S: student; C: course in
rule teaches (F, S) ⇔ instructs (F, C),
                      enrolls (S, C)
```

The signature of teaches is the binary relation of type faculty and student.

Finally in the third phase the user engages in an interactive dialogue. The user gives a query and the system responds with all the solutions. For example, the query to find all the students who are taught by the faculty member nancy is stated as follows:

```
let S : student in
  list S such that teaches (nancy, S)
```

The system responds with a list of the

students taught by the faculty nancy—the student john and the graduate student tim. The graduate student tim appears in the response because due to the fact that a graduate student is also a student. Now, if we wanted only the graduate students who are taught by nancy, we would write:

```
let GS : grad in
   list GS such that teaches(nancy, GS)
```

Here are two additional queries that illustrate some of the convenience of O^2 LDM.

The query

```
let S–student in
   list S.name such that teaches(tim.
   adviser, S)
```

finds the names of all the students who are taught by tim's adviser. We use field selection to denote tim's adviser. The query

```
let F : faculty; GS : grad in
   list GS.name such that teaches(F,
   GS), F.rank="asst"
```

finds the names of all the graduate students who are taught by assistant professors.

5. Comparison with PROLOG

In PROLOG [Clocksin 1984] the arguments to a relation can be atoms, variables, or terms made up of uninterpreted function symbols called functors. These functors behave as instantiated, partially instantiated, or uninstantiated record structures. This is a remarkably useful data structure considering its simplicity.

However, it is not ideal. The lack of type checking permits any structure to appear regardless of the relation. Furthermore, some natural notions are not easily expressed in this framework. We examine this issue in the context of the conceptual model for university life.

The PROLOG representation of the facts and rules in the university example is shown below:

```
instructs(faculty(peter, 2865, mis, prof),
          course(mis,600)).
instructs(faculty(nancy, 4354, cs, asst),
          course(cs,565)).
instructs(faculty(nancy, 4354, cs, asst),
          course(cs,502)).

enrolls(student(john, 8644, 5), course
        (cs,565)).
enrolls(student(john, 8644, 5), course
        (cs,502)).

enrolls(grad(tim, 2454, 6, mis,
faculty(peter, 2865, mis, prof)), course
        (mis,600)).
enrolls(grad(tim, 2454, 6, mis,

faculty(peter, 2865, mis, prof)), course
        (cs,565)).

teaches(F, S) : − instructs(F, C), en-
        rolls(S, C).
```

The query to determine all the students who are taught by the faculty member nancy is stated as follows:

```
?−teaches(faculty(nancy, −, −, −), S).
```

PROLOG finds the solutions to this query, namely:

```
student(john, 8644, 5)
student(john, 8644, 5)
grad(tim, 24548, 6, mis,
     faculty(peter, 2865, mis, prof))
```

We run into difficulty when we try to formulate the query to find the names of all the students (including graduate students) who are taught by nancy. The query:

```
?-
teaches(faculty(nancy, −, −, −), student
(X, −, −)).
```

fails to find the names of the graduate students. The problem arises because we have not captured the inheritance relation between graduate students and students. Taking inspiration from Zaniolo, we could introduce some new predicates that assert the "type" of the structures, as in:

```
is_student(student(NAME, ID, GPA)).
```

Then we could add rules to model the inheritance relation, as in:

```
is_student(student(NAME, ID, GPA)):−

is_grad(grad(NAME, ID, GPA, −, −)).
```

which states that "every graduate student is a student". But alone, this does not solve the problem, because we must still inject this inheritance relation into every relation. The following rule, however, does the trick:

```
enrolls(student(NAME, ID, GPA), C):−
enrolls(grad(NAME, ID, GPA, −, −), C).
```

The appropriate fields are dropped from grad to form the term for a student. However, the inheritance must be expressed this way for every relation. This is a problem when new relations are introduced. Furthermore, A "it-Kaci and Nasr observed that inheritance captured in this manner, i.e., through the use of logical implication, leads to a lengthening of proofs. They proposed a simple and efficient solution to the problem in the language LOGIN [Ait-Kaci].

The query to determine the names of all the graduate students who are taught by their adviser can be expressed as follows:

?-teaches(F, grad(NAME, -, -, -, -, F))

But we run into difficulty in trying to determine the names of the graduate students who are taught by tim's adviser. To understand the difficulty, consider the following query:

?-teaches(F, grad(NAME, -, -, -, -, -))

In the above query we need to bind the variable F to tim's adviser. To do this we need to determine some goal in which tim participates, and which we know will always be true. Since we know that tim is enrolled in a course, we could do the following:

?- enrolls(grad(tim, -, -, -, -, F), teaches(F, grad(NAME, -, -, -, -, -))

The above solution works because we know that there is a fact in which tim participates. But, this may not necessarily be true. The important point here is that the lack of expressions forces us to resort to obscure means to formulate the query. This is a fundamental limitation of PROLOG, one that has been overcome in O^2 LDM.

Ⅲ. Resolution in the presence of Subtyping

Here we concentrate on the aspects concerning the modification of the resolution algorithm required to handle expressions and subtypes. The prototype implementation of O^2LDM is written in ML[Milner 1990].

1. The Type-Object Lattice

In O^2LDM, the set of all types when ordered by the subtype relation forms a partial order. The partial order can be represented graphically as a lattice. We usually represent some finite portion of the subtype relation by a *Hasse diagram.*

A Hasse diagram is a graph structure whose nodes represent the type elements and whose edges are directed downward, from node τ to node η if η is covered by τ. A type η is covered by type τ, or τ covers η, if $\eta < \tau$ and there is no type γ such that $\eta < \gamma < \tau$. The symbols 《and》 are used to

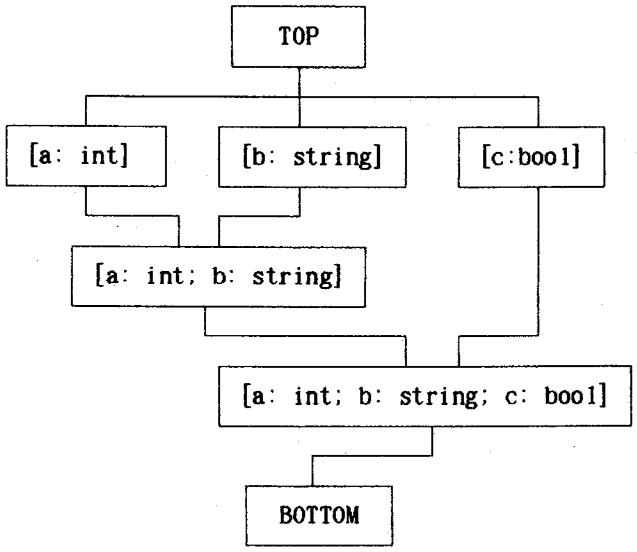


Figure 5 : Type Lattice

represent the relations covered and cover, respectively.

Figure 5 is a Hasse diagram of a sample portion of the type lattice. In Figure 5, [a: int; b: string; c: bool] 《[c: bool], because there is no type element γ in the lattice such that [a: int; b: string; c: bool] $< \gamma <$ [c: bool].

Given two types τ and in the lattice, the *meet* of τ and σ is the highest node in the type lattice η for which there is a path downward to η from both τ and σ. For example, in Figure 5, the meet of [a: int] and [b: string] is the type [a: int; b: string]. Similarly we can define the dual notion. The *join* of τ and σ is the lowest node in the type lattice γ, for which there is a path downward from γ to both τ and σ.

At the start of an interactive programming session in O^{2}LDM the universe of known objects is empty. The universe is filled with objects that are encountered when the user enters knowledge about a particular instance of the problem domain. As expressions are encountered they are evaluated and the resulting objects are added to the universe each according to its type. For example, when the user enters the following fact:

fact p(o_3, [a := o_1])

the expressions o_3 and [a := o_1] are

evaluated and three new objects are added to the universe—the object o_3, the object [a := o_1] and its subobject o_1.

The universe of objects determines the objects that can be bound to a logic variable used in a query. We call the data structure that is used to maintain the universe of objects the type-object lattice. The ML data types for the type-object lattice are given below. The type-object lattice for O^2 LDM is represented by the ML object TypeLattice. It initially contains only the single type Top.

```
( ******************
LATTICE NODE=Type*Mark*Exprs*
Subnodes
 ******************)

 datatype LATTICE=
   Lattice of TYPE * (bool ref) *
 (EXPR Set
   ref) * (LATTICE ref Set ref);

 type Subnodes=LATTICE ref Set;

( ******************
 Initialize the LATTICE
 by creating the Top Node.
 ******************)

 val TypeLattice=
   [ref (Lattice (Top, ref false, ref
 EmptySet,
          ref EmptySet))];
```

Algorithms for determining the existence of a type, adding a non-existent type, adding an object, and finding all objects of a type are ommitted because of the limited space.

2. Resolution

In O^2LDM, resolution is the rule of inference that is used to solve queries. The specific resolution strategy that is used in the language is a form of linear input resolution. This is the same strategy used by PROLOG. (See [Clocksin 1984, Gallier 1986] for more details about resolution theorem proving.)

The facts and rules that are entered into the O^2LDM system are the known clauses of the problem domain. A query in the language consists of a conjunction of goals to be satisfied. We start with the leftmost goal in the query and resolve it with one of the known clauses in the system to generate a new conjunction of goals. Then we resolve the leftmost goal of the new conjunction of goals to obtain another conjunction of goals, and so on. We continue this process until we are left with no more goals to be resolved, or until there are no clauses in the system that can be resolved with the chosen goal. If we are left with no more goals to be resolved then the query is satisfied, otherwise we have failed.

At each step, the clause that is chosen for resolution with the goal must satisfy the matching criteria—the head of the clause must match the goal under consideration. Once resolved, the goal is removed from the conjunction of goals and in its place the subgoals that make up the body of the matched clause are inserted. In other words, the subgoals that make up the body of the clause are appended to the front of the conjunction of subgoals. This means that O^2LDM finishes satisfying the newly added subgoals before it goes on to try something else.

If there is more than one clause that satisfies the matching criteria, O^2LDM considers one alternative at a time, fully exploring the alternative under the assumption that the choice is correct. The other alternatives are considered only after the chosen alternative has been fully explored. The clauses are considered in the order in which they are entered into the system.

3. Semantic Unification

It is the algorithm of matching literals that makes O^2LDM unique. We call the matching of two literals semantic unification. The result of semantic unification is a list of substitutions, not just a single substitution as usual in unification. Each substitution when applied to the two literals make them semantically equivalent.

The unification mechanism is different from PROLOG, where unification is strictly a syntactic process. The result of unifying two terms in PROLOG is a single substitution, not a list of substitutions. The resulting substitution, when applied to the two terms, makes them syntactically identical. For example, syntactic unification can unify the two terms $+(3, X)$ and $+(3,5)$, but not the terms $+(3,X)$ and $+(2,6)$ even though we know that if X were bound to 5 the two expressions would be semantically equivalent.

A *substitution* is a finite list of the form $\{(v\_1, \tau\_1)=e\_1, \cdots\cdots, (v\_n, \tau\_n)=e\_n\}$, where each element $(v\_i, \tau\_i)=e\_i$ is a binding of a logic variable named v_i, of type τ_i, to an expression e_i. For a given substitution, the logic variables in the substitution are distinct. The e_i's in the bindings are restricted to ground expressions and logic variables. For the sake of brevity, the type of the logic variable will be omitted from the substitution in some contexts.

Based on the form of the expressions p and q, the semantic unification algorithm is divided into the following four cases:

- *p* and *q* are both logic variables.
- *p* is a logic variable, but *q* is any expression other than a logic variable.
- *q* is a logic variable, but *p* is any expression other than a logic variable.
- Both *p* and *q* are expressions, but neither is a logic variable.

Next we describe the individual cases in detail.

Case 1 :

Let *p* be a logic variable ($p_$v, τ), where $p_$v is the name of the variable and τ its type, and *q* a logic variable ($q_$v, σ), where $q_$v is the name and the type.

If both p and q are of the same type, i.e., $\tau=\sigma$, then any object from the domain of the logic variable *p* (which is also the domain of *q*) can be bound to both variables making them semantically equivalent. The substitution that would make *p* and *q* semantically equivalent is one that contains the binding ($p_$v, τ)=($q_$v, σ). Thus the new substitution, S_n, is obtained by adding this new binding to the current substitution, S.

If $\tau \neq \sigma$, then the only objects that can be bound to both variables are objects from the intersecting domain. The intersecting domain is all the objects in the type-object lattice of type δ, where δ is the meet of τ and σ. The new substitution S_n is obtained by adding to the current substitution S, the bindings ($p_$v, τ)=($q_$v, δ) and ($q_$v, σ)=($q_$v, σ). The reason for including the binding ($q_$v, σ)=($q_$v, δ) is because the type of the logic variable *q* has been constrained to a subtype of its original type, as a result of unifying p and *q*. If δ=Bottom, then unification fails, because there are no objects of type Bottom in the lattice.

Case 2:

In this case *p* is a logic variable and *q* is not. Let p be the logic variable ($p_$v, τ). The expression *q* may contain free logic variables. Let *F* represent the set of free variables in *q*. There are two potential cases to be considered.

In the first case, $F=\emptyset$, *i.e.*, *q* has no free variables. For *p* and *q* to be unifiable the type of *q* must be a subtype of the logic variable p. The new substitution, S_n, is obtained by adding the binding $(p\_v, \tau)=q$ to the current substitution S. The result of semantic unification in this case is a single substitution.

Next, we consider the case when $F \neq \emptyset$, i.e., q contains free variables. Since a free variable is a variable that is not bound to anything in S, it can be instantiated to any object from its domain. The domain of a free variable is all the objects in the type–object lattice of type τ, where τ is the type of the free variable.

For p and q to be unifiable: (1) the free variables in q must be bound to objects from their respective domains and (2) the type of object o must be a subtype of the type of the logic variable p, where o is the result of evaluating the expression q after all its free variables have been instantiated. The new substitution is obtained by adding the binding $(p\_v, \tau)=o$ to S and S_f, where S is the current substitution, and S_f the binding of free variables to objects.

There is a list of these new substitutions that make p and q unifiable—potentially as many as all possible combinations of free variable–object bindings. For example, let us assume that the expression q contains two free variables—U and V. Let the domain of U contain the objects o_1 and o_2, and the domain of V the objects o_3 and o_4. The possible combinations of free variable–object bindings are: {U=o_1, V=o_3}, {U=o_1, V=o_4}, {U=o_2, V=o_3}, {U=o_2, V=o_4}.

Let F be the free variable set

{(v_1, τ_1), ……, (v_i, τ_i), ……, (v_n, τ_n)}

where each (v_i, τ_i) is a free variable, v_i the name of the free variable and τ_i its type. Let the domain of the free variables be represented by D(τ_i) for all i∈1, ……, n.

Let S_j={(v_1, τ_1)=o_1, ……, (v_n, τ_n)=o_n}

where $o\_1 \in D(\tau\_i)$, ……, $o\_n \in D(\tau\_n)$ represent the substitution obtained by binding the free variables to objects from their respective domains. We can obtain a list of such substitutions—one for each combination of free variable–object bindings. The list of such substitutions will be denoted by S_l. From S_l we determine the new substitution list as follows.

Let S_j be an element of S_l, and o_j the object that results from evaluating the expression q using the substitution S_j. Let F be a function that, given S–j, returns the pair (o_j,S_j). Let G be a function that returns true if the type of o_j is a subtype of the type of the logic variable p, otherwise it returns false. Let H be a function that, given the pair (o_j,S_j), returns the substitution obtained by adding the binding (p_v, τ)=o_j to S and S_j, where S is the current substitution. The list of new substitutions is obtained by:

(map H) ∘ (filter G) ∘ (map F) S_l

where ∘ denotes the composition of functions, map and filter represent the higher-order functions with their usual meaning.

Case 3:

This case is similar to case 2.

Case 4:

In this case neither p nor q is a logic variable. Both expressions, p and q, may contain free variables. Let F_p represent the free variables in p, and F_q the free variables in q. Let F represent ($F\_p \cup F\text{-}q$).

Let us first consider the case when $F=\emptyset$, i.e., both p and q have no free variables. For p and q to be unifiable the two expressions p and q must be identical. The new substitution, S_n, is identical to the current substitution S. Since there are no free variables in p and q, no new bindings are formed.

Next we consider the case when $F \neq \emptyset$, i.e., either p or q, or both contain free variables. Since a free variable is a variable that is not bound to anything in S, it can be instantiated to any object from its domain. For p and q to be unifiable: (1) the free variables in p and q must be bound to objects from their respective domains (2) the objects o_p and o_q must be identical, where o_p and o_q are the objects that result from evaluating p and q respectively, after all their free variables have been instantiated. The new substitution is S and S_f, where S is the current substitution, and S_f the binding of free variables to objects.

There is a list of these new substitutions that make p and q unifiable-potentially as many as all possible combinations of free variable-object bindings. To determine this list of new substitutions, we need S_l, the list of all possible combinations of free variable-object bindings. From S_l we determine the new substitution list as follows.

Let S_j be an element of S_l, and $\hat{o}$j_p and $\hat{o}$j_q the objects that result from evaluating the expressions p and q, respectively, using the substitution S_j. Let F be a function that, given S_j, returns the triple ($\hat{o}$j_p, $\hat{o}$j_q, S_j). Let G be a function that returns true if the objects $\hat{o}$j_p and $\hat{o}j\_q$ are identical, otherwise it returns false. Let H be a function that, given the triple ($\hat{o}j\_p$, $\hat{o}j\_q$, S_j), returns the substitution S and S_j, where S is the current substitution. The list of new substitutions is obtained by:

$$(\text{map H}) \circ (\text{filter G}) \circ (\text{map F})\ S\_1$$

This completes the description of the algorithm.

4. Examples

We look at two examples to illustrate the deductive process using semantic unification. The first example shows the importance of the semantic unification, the second example shows the importance of subtyping.

(1) An example of semantic unification

In this example we trace through the execution of the query q_4 (M,N). The universe of objects is shown in Figure 6. The rules and facts of the example are shown in Figure 7 along with the entire search space of the query in Figure 8. In the search space any path from the original goal to a box with the caption "success" represents a solution. Ultimately, the query yields two substitutions for the variables M and N as solutions. These two solutions are:

{M=[a:=o_1], N=o_3}
{M=[a:=o_2], N=o_3}

The way O^2LDM finds these solutions is as follows. O^2LDM searches through the list of clauses and finds that the goal q_4 (M, N) unifies the head of the forth clause, q_4(P, Q.b). The result of unifying the two literals, q_4(M, N) and q_4(P, Q. b), is two substitutions. O^2LDM determines the two substitutions as follows.

For the literals, q_4(M, N) and q_4(P, Q.b), to match their predicates must be identical and their respective arguments must match, i.e., they must be semantically equivalent. The substitution {M=P} makes the first arguments, M and P, semantically equivalent. Since Q is not bound to anything in the current substitution, {M=P}, we need to find objects in the domain of Q that when bound to Q will render the two expressions Q.b and N semantically equivalent.

What is the domain of the logic variable Q? What objects can Q be bound to? Clearly, if we are to consider a universe of all possible values, including all integers, strings, records, etc., we could have an infinite number of objects of type [b:β], and thus an infinite number of substitutions

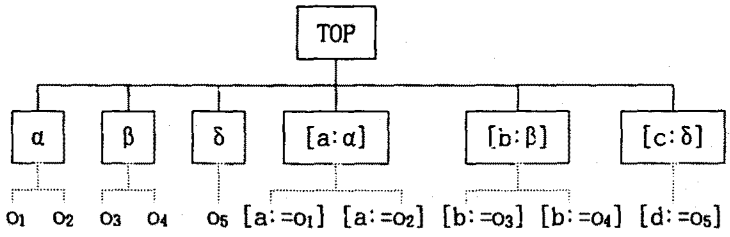


Figure 6 : Universe of objects

```
signature q_1([b:β],[d:δ])
signature q_2(α,[b:β])

fact q_1([b:=o_3],[d:=o_5])
fact q_2(o_1,[b:=o_3])
fact q_2(o_2,[b:=o_3])

let P:[a:α],Q: [b:β] in
  rule q_4(P, Q.b)
         <=> q_1(Q, [d:=o_5]), q_2(P.a, Q)
```

Figure 7 : O²LDM program

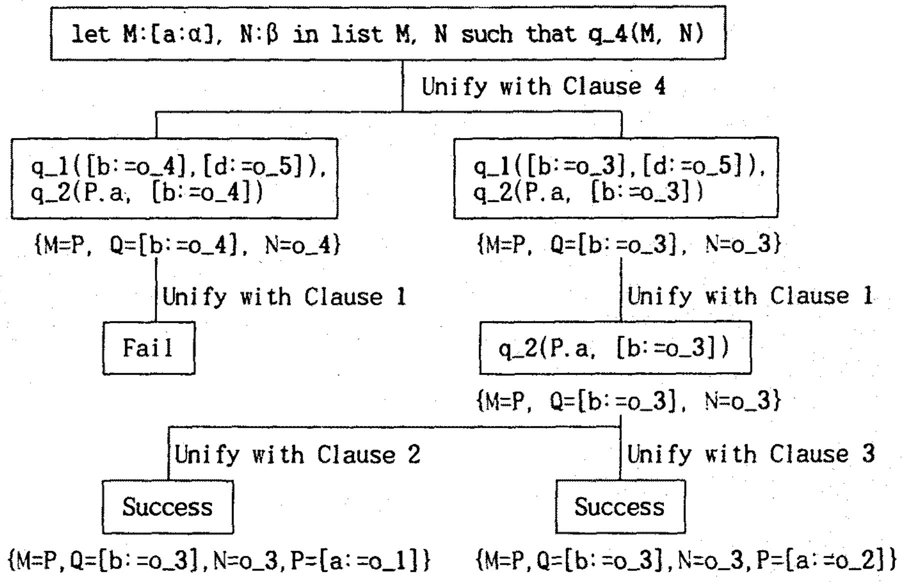


Figure 8 : O²LDM query and search space

that make N and Q.b semantically equivalent. But, in O^{2}LDM, the universe consists of only the objects that have been introduced to the system by a particular point in time.

There are two objects in the domain of Q. These are the two objects of type [b:β]. Each object when bound to Q makes the two expressions Q.b and N semantically equivalent. The corresponding binding to N will be the field component labeled b of the object that is bound to Q. The result of unifying q_4(M, N) and q_4(P, Q.b) is the following two substitutions:

{M=P, Q=[b:=o_3], N=o_3}
{M=P, Q=[b:=o_4], N=o_4}

We can choose either of the two substitutions for the next step. If we reach a dead end by choosing one, we can pursue the other. As a result of unification we have a fan-out (branching) in the search space of Figure 7. The degree of the fan-out is the number of substitutions returned. The result of a successful unification in PROLOG is a single substitution.

Let us choose the substitution:

{M=P, Q=[b:=o_4], N=o_4}

for the next step. The goal q_4(M, N) is replaced by the body of clause 4, and appropriate substitutions are made, yielding the following:

q1([b:=o_4], [d:=o_5]), q2(P.a, [b:=o_4])

Next we try matching the goal q1([b:=o_4], [d:=o_5]). The goal does not unify with the head of any of the clauses in our list of clauses. It does not match with first clause, because [b:=o_3] and [b:=o_4] are two different objects and hence are not semantically equivalent. We have reached a dead end. In the search space, boxes with the caption "fail" denote failures that result because one or more of the arguments of the literals did not match, even though the predicates matched.

We backtrack and pursue the other substitution:

{M=P, Q=[b:=o_3], N=o_3}

The goal q_4(M,N) is replaced by the body of clause 4 and appropriate substitutions are made yielding:

q1([b:=o_3], [d:=o_5]), q2(P.a, [b:=o_3])}

The goal q1([b:=o_3], [d:=o_5]) successfully unifies with the head of the first clause. No new bindings are introduced. As a result the substitution remains the same. Notice that the degree of fan-out in the search space at this point is one. Since the body of the first clause is empty, our new goal is:

q2(P.a, [b:=o_3])

We try matching the goal q2(P.a, [b:=o_3]), with the head of the second clause, q2(o_1, [b:=o_3]). Since P is not bound to anything in the current substitution, we need to find objects in the domain of P that when bound to P will render the two expression P.a and o_1 semantically equivalent. There are two objects in the domain of P—the two objects in the universe of type [a:α]. Of the two objects only the object [a:=o_1] when bound to P makes the two expressions P.a and o_1 semantically equivalent. We end up with the following substitution:

{M=P, Q=[b:=o_3], N=o_3,P=[a:=o_1]}

Since the body of the second clause is empty we are left with no more goals. An empty list of goals indicates success. The substitution for M and N that makes q_4 (M, N) true is:

{M=[a:=o_1], N=o_3}

The other solution can be found by backtracking.

(2) An example of semantic unification with subtyping

We extend the example of the previous section to illustrate inheritance. Consider the universe shown in Figure 9 and the search space in Figure 10. We trace through the execution of the query q_4(M, N). Note that the logic variable M is not of type [a:α], but a subtype of it. The type of M is [a :α ; e :σ]. The entire search space of the query q_4(M, N) is shown in Figure 10. The query yields only one substitution for M and N as solution:

{M=[a :=o_1; e :=o_6], N=o_3}

O^2LDM finds the above solution as follows. The goal q_4(M, N) unifies with the head of the fourth clause q_4(P, Q.b) yielding the following substitutions:

{M[a:α ; e:σ]=P[a:α ;e:σ],
P[a:α]=P[a:α ;e:σ],
Q=[b:=o_3], N=o_3}
{M[a:α ;e:σ]=P[a:α ;e:σ],
P[a:α]=P[a:α ;e:σ],
Q=[b:=o_4], N=o_4}

It is important to compare the above substitutions with their counterparts in the previous example. Notice that besides type

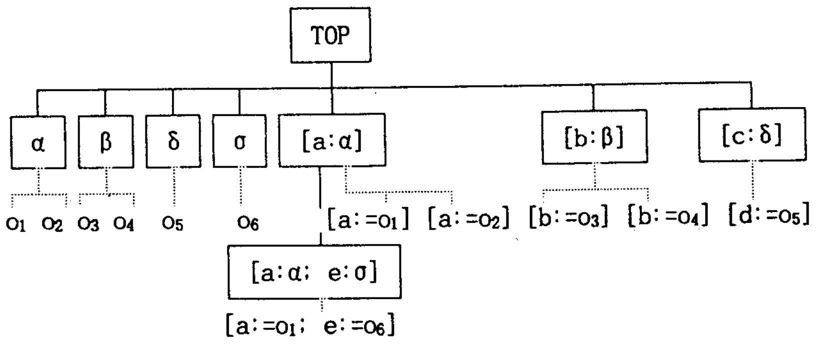
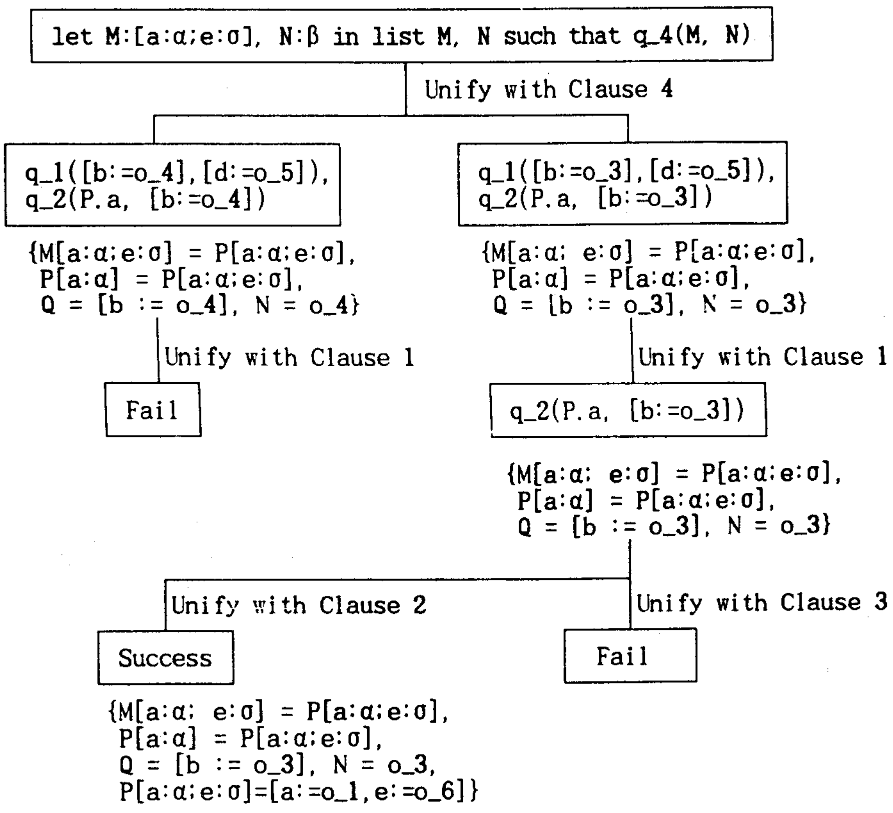


Figure 9 : Expanded Universe of objects



let M:[a:α;e:σ], N:β in list M, N such that q_4(M, N)
Unify with Clause 4
q_1([b:=o_4],[d:=o_5]), q_2(P.a, [b:=o_4])
{M[a:α;e:σ] = P[a:α;e:σ], P[a:α] = P[a:α;e:σ], Q = [b := o_4], N = o_4}
q_1([b:=o_3],[d:=o_5]), q_2(P.a, [b:=o_3])
{M[a:α; e:σ] = P[a:α;e:σ], P[a:α] = P[a:α;e:σ], Q = [b := o_3], N = o_3}
Unify with Clause 1
Fail
Unify with Clause 1
q_2(P.a, [b:=o_3])
{M[a:α: e:σ] = P[a:α;e:σ], P[a:α] = P[a:α;e:σ], Q = [b := o_3], N = o_3}
Unify with Clause 2
Unify with Clause 3
Success
Fail
{M[a:α; e:σ] = P[a:α;e:σ], P[a:α] = P[a:α;e:σ], Q = [b := o_3], N = o_3, P[a:α;e:σ]=[a:=o_1,e:=o_6]}

Figure 10 : O²LDM query and search space

information there is also an additional binding in each of the above substitutions. The additional binding is P[a:α]=P[a:α ; e:σ]. The reason for this additional binding is that since the logic variable M is of type [a:α ;e:σ], unifying M and P has constrained P to a subtype of its original type. The type of P for all future references is [a:α ;e:σ].

As before we can pursue either substitution for the next step. If we choose the second substitution, replace the goal q_4(M, N) by the body of clause 4, and make the appropriate substitutions we end up with the following:

{q1([b:=o_4], [d:=o_5]), q2(P.a, [b:=o_4])}

Next, we try matching q1([b:=o_4], [d:=o_5]). This goal does not unify with the head of any of the clauses in the list of clauses. We have reached a dead end, so we backtrack and pursue the other substitution:

{M[a:α ; e:σ]=P[a:α ; e:σ],
P[a:α]=P[a:α ; e:σ],
Q=[b:=o_3], N=o_3}

and obtain the following goals:

q1([b:=o_3], [d:=o_5]), q2(P.a, [b:=o_3])

This time the goal q1([b:=o_3], [d:=o_5]) matches with the head of the first clause. No new bindings are introduced as a result of the match, and since the body is empty we are left with:

q2(P.a, [b:=o_3])

as the next goal. We try to match q2(P.a, [b:=o_3]), with the head of the second clause q2(o_1, [b:=o_3]). Since P is not bound to anything in the current substitution, we need to find objects in the domain of P that when bound to P will render the two expressions P.a and o_1 semantically equivalent. The type of P is [a:α ; e:σ], not [a:α]. There is only one object in the universe of type [a:α ; e:σ]. The object when bound to P makes the expressions P.a and o_1 semantically equivalent. The new substitution is:

{M[a:α ;e:σ]=P[a:α ;e:σ],
P[a:α]=P[a:α ;e:σ],
Q=[b:=o_3], N=o_3,
P=[a:=o_1; e:=o_6]}

We are then left with no more goals. The substitution for M and N that makes q_4(M,N) true is:

{M=[a:=o_1; e:=o_6], N=o_3}

Ⅳ. Conclusion

The primary objective of this research was to develop a programming language in which knowledge about a domain can be safely and concisely expressed. Domain knowledge has many forms, and in order to express these diverse forms of knowledge, we need a multiparadigm language. Some forms of knowledge can be stated more naturally and concisely in one paradigm than in another.

Several other researchers have also argued for the need to integrate the paradigms and numerous multiparadigm environments were built. We do not embed one or more paradigms on top of another. Instead we identify the primitive language features that are necessary for representing knowledge about a domain and then introduce these features in a new language. We were interested in studying the semantics of combining the paradigms.

We are also convinced that compile–time type checking is an important language design goal. We feel that support for a paradigm in a language comes not only in the obvious form of language primitives that allow programming in the paradigm, but also in the more subtle form of type checking to ensure that unintentional deviations are detected early.

O^2LDM provides powerful primitives for representing domain knowledge — domain objects and their attributes, the taxonomic arrangement of domain objects, the association of one or more objects in a relationship, and rules for making decisions. But more importantly it provides a reasonably coherent semantics of the combination of these primitives.

The focus of the development of O^2LDM has been in its deductive process, not in developing it into a practical language. Some of the possible directions in which the language can be extended are outlined below.

1) The incorporation of type constructors such as cartesian products, lists, etc., would make the language more convenient to program in. The results of queries, namely lists, could be treated as objects in O^2LDM.

2) The use of lazy lists could be used as the appropriate data structure to manage queries with an infinite number of solutions.

3) Imperative features are needed in the language to model state transformation in the domain. We need operators for changing the set of clauses in the knowledge base and a type–object lattice that

could change during query.

4) Another extension would be a default mechanism capability. Presently there are no mechanisms for objects to inherit default values when they are created.

5) Domain objects and relationships should be persistent. This would support the use of O^2LDM as a database language.

6) Another possibility would be to add transaction management, moving O^2LDM in the direction of Galileo.

참 고 문 헌

[1] A" it-Kaci, H. and Nasr, R. "LOGIN: A Logic Programming Language with built-in Inheritance," *Journal of Logic Programming*, No. 3, pp. 185-215, 1986.

[2] Albano, A., Cardelli, L. and Orsini, R. "Galileo: A Strongly-Typed, Interactive Programming Language," *ACM Transactions on Database Systems*, Vol. 10, No. 2, pp. 230-260, June 1985.

[3] Atkinson, M.P. and Buneman, O.P. "Types and persistence in database programming languages." *ACM Computng Surveys*, Vol. 19, No. 2, pp. 105-190, June 1987.

[4] Bobrow, D.G. and Stefik, M. *The LOOPS Manual*, Xerox PARC, 1983.

[5] Bobrow, D.G. "If Prolog is the answer, what is the question? Or what it takes to support AI programming paradigms," *IEEE Trans. on Software Eng.*, Vol. SE-11, No. 11, pp. 1401-1408, November 1985.

[6] Borgida, A. "Features of languages for the development of information systems at the conceptual level," *IEEE Software*, Vol.2, No. 1, pp. 63-72, January 1985.

[7] Cardelli, L. "A semantics of multiple inheritance," *Information and Computation*, Vol. 76, pp. 138-164, 1988.

[8] Clocksin, W.F. and Mellish, C.S. *Programming in Prolog*. Berlin: Springer-Verlag, 1984.

[9] Fikes, R. and Kehler, T. "The role of frame-based representation in reasoning," *Communications of ACM*, Vol. 28, No. 9, pp. 904-920, September 1985.

[10] Gallier, J.H. *Logic for Computer Science:*

Foundations of Automatic Theorem Proving, New York: Harper & Row, 1986.

[11] Goguen, J.A. and Meseguer, J. "Unifying functional, object-oriented and relational programming with logical semantics," *Research Directions in Object-oriented Programming*, edited by P. Wegner and B. D. Shriver. Cambridge, MA: MIT Press, pp. 417–477, 1987.

[12] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Reading: Addison-Wesley, 1983.

[13] Gullichsen, E. *BiggerTalk: Object-Oriented PROLOG*, Technical Report STP-125-85, MCC Software Technology Program, Austin, Texas, 1985.

[14] Kifer, M. and Lausen, G. "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme," *Proceedings of the ACM SIGMOD Conference*, pp. 134–146, 1989.

[15] Milner, R., Tofte, M. and Harper, R. *The Definition of Standard ML*. Cambridge, MA: MIT Press, 1990.

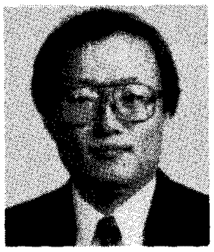
[16] Ohori, A., Buneman, P. and Breazu-Tannen, V. "Database programming in Machiavelli-A polymorphic language with static type inference," *Proceedings of the ACM SIGMOD Conference*, pp. 46–57, 1989.

[17] Ohori, A. and Buneman, P. "Static type inference for parametric classes," *Object-oriented Programming: Systems, Languages and Applications*, edited by Norman Meyrowitz. New York: ACM, 1989.

[18] Stansifer, R., Jung, C., Whinston, A. and Bhasker, P. *Description of SEMLOG*. CSD-TR-868, Department of Computer Sciences, Purdue University.

[19] Zaniolo, C. "Object-Oriented Programming in PROLOG," *Proceedings of the 1984 IEEE Symposium on Logic Programming*, pp. 265–270, 1984.

◇ 저자소개 ◇



저자 **정철용**은 현재 상명여자대학교 경영학과 조교수로 재직중이다. 그는 서울대학교 경제학과에서 학사 학위, 미국 University of Washington의 경영대학에서 경영학 석사 학위, 그리고 University of Texas at Austin에서 경영정보학 박사학위를 받았다. University of Texas at Austin에서 강사로 재직하였으며 한국금융연구원 부연구위원 을 역임하였다. 주요 연구분야는 인공지능의 데이타베이스 시스템에의 응용이며, 자산 부채종합관리시스템 등 금융정보시스템 관련 연구과제를 수행중이다.