A Logic for Objects

David Maier

Technical Report CS/E-86-012
5 November 1986

Oregon Graduate Center
19600 S.W. von Neumann Drive
Beaverton, Oregon 97006-1999

Presented at the Workshop of Deductive Databases and Logic Programming,
August 1986

Revised November 1986

A Logic for Objects

by David Maier

Abstract

This report presents a logic based on terms that represent the existence and internal structure of objects, rather than names of objects and relationships among those names. The logic incorporates a hierarchy to organize classes of objects, and structures in this logic are amenable to database representation. Updates in the logic preserve identity of entities, and provide the foundation for the semantics of rules, which are treated as deferred updates. The paper concludes by pointing out some challenges facing the top-down evaluation of rules, which will be the next area for continued research.

A Logic for Objects

David Maier
Dept. of Computer Science & Engineering
Oregon Graduate Center

What follows is an initial attempt at a formal basis for a combination of logic and object-oriented programming as it applies to databases. I want non-procedural querying from logic programming, along with a uniform language for query, update, virtual data definitions and constraints. From object-oriented programming, I want complex objects that can change their internal structure, object identity [KC] to give a handle on update, and a classification hierarchy for organizing the database scheme. I expect databases founded on this logic to store an explicit model, or a portion of one, as opposed to storing a theory and using an inferred model, as in Prolog. I believe Goguen and Meseguer are getting at the same thing in FOOPs with their semantics based on initial models [GM]. It also has some flavor of Montague's intensional logic [Mo], in that object identity is much like a constant intensional concept; it can track the indentity of entities through various worlds. Moffat and Gray [MG] show a means to bring objects into logic, but the semantics of the objects is outside the logic of the system.

1. Model Theory for Objects

Traditional logic was developed to give precise meaning to statements in natural language. It supports terms to capture noun phrases, literals to capture simple sentences using the noun phrases, and formulas to capture compound statements. It abstracts language more than it abstracts the structure of the real-world entities the language describes. One reason update is so clumsy in logic-based systems is that they must be cast as changes in statements about the world, rather than as changes in the structure of the world.

Example 1: With Horn clause logic programming, two facts cannot share subparts. During runtime, variables in different clauses can become linked, but no such linkage can be stored in the database. Thus, when two facts are talking about the same object in the world, they duplicate information about that object, and both must be updated if that object changes. If I have

```
dept("Sales", manager(name("Joe", "Doe"), address("204 SW 5th")).
dept("Marketing", manager(name("Joe", "Doe"), address("204 SW 5th")).
```

both talking about the same individual, and he moves, I must update both facts for the one change. Furthermore, the change is made by retracting those facts and asserting new facts

```
dept("Sales", manager(name("Joe", "Doe"), address("116 NW 6th")).
dept("Marketing", manager(name("Joe", "Doe"), address("116 NW 6th")).
```

In interpreting these new facts, nothing about them requires that the domain element that represents **name("Joe", "Doe")** in the first case is the same as in the second. I have no way of saying "Everything stayed the same except the address." While Prolog can be viewed as talking about a model—the minimum Herbrand model—that model assumes "the name's the thing": different names mean different things and different things cannot have the same description. Kowalski [Ko] maintains the connection between states by representing information as relationships that are initiated and terminated by events explicitly associated with them.

A logician might question the need for identity, citing Liebnitz's principle of "identity of indiscernables": Any two objects x and y are identical if and only if for every predicate P, $P(x) \iff P(y)$. That is, if no predicate can tell the difference between x and y, then they are the same object. Or, if two objects are not identical, some predicate must distinguish them. As MacLennan has pointed out [Ma], while identity of indiscernables might be a reasonable view of the real world, it is a dubious proposition for data models, because a database never abstracts everything about the world. It is useful to represent in a database that x and y are distinguishable, without having to find or make up a property to distinguish them. Furthermore, the property that distinguished them might not be part of their internal state, but have to do with relationships in which they participate: two 47Ω, ¼-watt resistors in different places in a circuit.

In the logic I present, there will not be a duality between terms and literals. All terms will be construed as assertions of existence of objects in the world, and relationships among them. A

database in the object system admits interpretation as a particular model of the logic (and *for* the world) or certain aspects of the database semantics can be ignored, to view it as a theory. The former interpretation will be most valuable for update, while the latter view better supports the idea of rules for virtual objects and relationships.

1.1. O-Terms

Here I present *O-terms* (for *object* terms), which are similar in syntax, if not meaning, to the ψ-terms of Ait-Kaci [A-K]. O-terms will be built up recursively from a set D of basic terms called ***data values***. The set of data values can be thought of as names of all objects with no further internal structure. For exposition purposes, I will let D be integers and strings. O-terms need not mention any data values at all, as they can also bottom out in ***object variables***, denoted by capital letters. I also assume a set L of *labels* or field names to build terms, which I shall consider disjoint from D.

An O-term is either

(1) A data value $d \in D$,

(2) An object variable **V**, or

(3) A complex O-term

$$\mathtt{X(lab1} \longrightarrow t_1, \ \ldots, \ \mathtt{labn} \longrightarrow t_n\mathtt{)}$$

where **X** is an object variable, **lab1, ..., labn** are in L and $t_1, \ldots, t_n$ are O-terms.

I will sometimes call **lab** $\longrightarrow t$ a *field*, and t the *field value*.

Example 2: The following are all legal O-terms.

```
E(name → N(first → "Joe", last → "Doe"),
  degree → D(level → "PhD", year → 1981,
              school → S(name → "SMU", state → "TX")))

E(name → N(last → L),
  degree → D(school → S(name → L)))

E

"Joe"
```

In general, object variables will only be important when repeated, and we can omit non-repeated variables from an O-term for simplicity.

1.2. O-Logic

O-logic formulas are built up from O-terms, logical connectives ($\wedge \vee \neg \Rightarrow$) and quantifiers ($\forall$ $\exists$) on object variables, in the same way as predicate logic formulas are built up from atomic formulas.

Example 3: An O-logic formula f =

```
∀ D ∀ M D(manager → M) ⇒
    M(worksIn → D) ∨ ∃C M(worksIn → C(dname → "Admin"))
```

Informal translation: A manager of a department works in the department or works in a department named **"Admin"**.

1.3. Object Models

Given an O-logic formula f, what is a model for f? I will first define structures, and then define when a formula is true for a structure. A *structure* ST for f is a pair $<U, g>$. U is the *universe*, which is assumed non-empty. It consists of D, the set of data values, plus W, a disjoint set of *entities*. The universe is the set of all objects known to exist. The second component of the structure, g, interprets labels as functions. It maps labels in L to partial functions from W to U. Thus, g(**worksIn**) will be a partial function from entities to entities and data values.

Example 4: For the formula f above, a structure is $ST = <U, g>$ where

$$U = \{e_1, e_2, e_3, e_4, e_5\} \cup D$$

and

$$g(\mathbf{manager})(e_1) = e_2,\ g(\mathbf{manager})(e_3) = e_4,\ g(\mathbf{worksIn})(e_2) = e_1,\ g(\mathbf{worksIn})(e_4) = e_5,$$

$$g(\mathbf{dname})(e_1) = \texttt{"Sales"},\ g(\mathbf{dname})(e_3) = \texttt{"Manuf"},\ g(\mathbf{dname})(e_5) = \texttt{"RandD"}.$$

(Intuitively, e_1, e_3 and e_5 are departments, and e_2 and e_4 are employees, but nothing in the model states this condition. I will be able to impose such a constraint once I add a class hierarchy. Again, intuitively, this structure does not satisfy the formula f, because e_4 manages **"Manuf"** but works in **"RandD"**.) O-logic structures talk about the internal compositions of entities, where usual logic structures talk about relationships among objects that have no internal state.

Before defining the meaning of a formula in a structure, I need a mapping from unbound variables to elements of the universe. Such a mapping will allow me to define the meaning function recursively in the presence of quantifiers. It will also allow me to assign meanings to open formulas. An *instantiation* I is a mapping from object variables to U. I extend I to terms by letting it be the identity on data values and the value of the head variable for complex terms. That is, for an O-term t, $I(t)$ is

(1) $I(\mathbf{Y})$ if $t = \mathbf{Y}$;

(2) d if $t = d \in D$;

(3) $I(\mathbf{Y})$ if $t = \mathbf{Y}(\ldots)$.

I define a *meaning function* $M_{ST,I}$ relative to a structure ST and and instantiation I. Start with the meaning of O-terms. For $d \in D$,

$$M_{ST,I}(d) = \text{true. (Object } I(d) = d \text{ exists.)}$$

For any variable $\mathbf{X}$,

$$M_{ST,I}(\mathbf{X}) = \text{true. (Object } I(\mathbf{X}) \text{ exists.)}$$

For a complex term t =

$$\texttt{X(lab1} \longrightarrow t_1\texttt{, ..., labn} \longrightarrow t_n\texttt{)}$$

define $M_{ST,I}(t)$ = true if

(1) $I(t)$ (= $I(\mathbf{X})$) is an entity, not a data value. (Only entities have internal state.)

(2) For $e = I(t)$ and $c_i = I(t_i)$, $1 \leq i \leq n$

$$g(\mathbf{labi})(e) = c_i,$$

so, in particular, $g(\mathbf{labi})$ is defined on e. ($I(\mathbf{X})$ has an internal state consistent with the term. Entity **X** may have more state than described by the term, however.)

(3) $M_{ST,I}(t_i)$ = true. (Sub-entities exist and have the required internal state.)

So the term t is true if the values in its fields and subfields correspond to those given by g, and the values in those fields are true.

Example 5: Let I_1 be the instantiation where $I_1(\mathbf{D}) = e_1$ and $I_1(\mathbf{M}) = e_2$. Using the structure ST from Example 4, for t =

$$\texttt{D(manager} \longrightarrow \texttt{M)}$$

$M_{ST,I_1}(t)$ is true, since $g(\mathbf{manager})(e_1) = e_2$.

For instantiation I_2, where $I_2(\mathbf{D}) = e_1$ and $I_2(\mathbf{M}) = e_3$, $M_{ST,I_2}(t)$ is false, because $g(\mathbf{manager})(e_1) \neq e_3$.

$M_{ST,I}(f \wedge g)$, $M_{ST,I}(f \vee g)$, $M_{ST,I}(f \Rightarrow g)$, and $M_{ST,I}(\neg f)$ are all defined in the obvious way. For quantifiers, I have

$M_{ST,I}(\forall\ \mathbf{V}\ f)$ = true if for every I_1 that agrees with I except on (possibly) $\mathbf{V}$, $M_{ST,I_1}(f)$ = true.

$M_{ST,I}(\exists\ \mathbf{V}\ f)$ = true if for some I_1 that agrees with I except on (possibly) $\mathbf{V}$, $M_{ST,I_1}(f)$ = true.

Note that $M_{ST,I}(f)$ does not depend on I if f is a closed formula. If f is closed and $M_{ST}(f)$ is true, then ST is a *model* for f.

Example 6: Consider $M_{ST}(f)$ for ST = $<U, g>$ and f as given in Examples 4 and 5.

$\forall$ `D` $\forall$ `M D(manager` $\longrightarrow$ `M)` $\Rightarrow$
    `M(worksIn` $\longrightarrow$ `D)` $\vee$ $\exists$`C M(worksIn` $\longrightarrow$ `C(dname` $\longrightarrow$ `"Admin"))`

$U = \{e_1, e_2, e_3, e_4, e_5\} \cup D$

g(**manager**)$(e_1) = e_2$, g(**manager**)$(e_3) = e_4$, g(**worksIn**)$(e_2) = e_1$, g(**worksIn**)$(e_4) = e_5$,

g(**dname**)$(e_1) =$ **"Sales"**, g(**dname**)$(e_3) =$ **"Manuf"**, g(**dname**)$(e_5) =$ **"RandD"**.

For instantiation I where I(**D**) $= e_3$ and I(**M**) $= e_4$, I have

$M_{ST,I}$(`D(manager` $\longrightarrow$ `M)`) = true, and

$M_{ST,I}$(`M(worksIn` $\longrightarrow$ `D)`) = false.

For the formula to be true, there must be an instantiation I_1 that agrees with I on **D** and **M** such that

M_{ST,I_1}(`M(worksIn` $\longrightarrow$ `C(dname` $\longrightarrow$ `"Admin")))` = true.

For g(**worksIn**)$(I_1$(**M**)) to equal I_1(**C**), I_1(**C**) must be e_5, since I_1(**M**) = I(**M**) = e_4. However, g(**dname**)$(e_5) =$ **"RandD"**. Thus, $M_{ST}(f)$ is false.

The plan is to store a structure as a database. While I assume D is regular enough to be described finitely, I do not plan to handle structures where W is infinite. Let an *entity finite* structure be one in which the set of entities is finite and the set of labels for which g is not the completely undefined function is finite. (That is, g has a finite set of "interesting values.") I can store an entity finite structure. Further, I can decide if a closed formula f is true in the structure. The only problem with applying the meaning function to f is the quantifiers. However, for any term **X** or **X(...)**, the term takes on a uniform value for all $d \in D$. Terms of the first sort are true for all elements of D, terms of the second sort are false. Thus a quantifier can by evaluated by considering all entities in W and one representative from D. Actually, the

situation is not that simple. Clearly, considering one data value from D for **X** in

$$\exists \texttt{Y} \ \exists \texttt{X} \ \texttt{Y(dname} \longrightarrow \texttt{X)}$$

is not sufficient. In such cases, however, a finite set of feasible bindings can be derived from the binding of the object variable for the enclosing term. Another approach I think works is to let bound variables range over W plus those values from D in the range of g plus one value from D not in the range of g.

Implication is defined in the usual manner. Formula f logically implies formula f' if for all structures ST and all instantiations I,

$M_{ST,I}(f') =$ true means $M_{ST,I}(f) =$ true.

If f and f' are closed, this just says the models of f are also models of f'.

Note that my interpretation of an O-logic term is different from that given by Ait-Kaci and Nasr [A-KN]. They map variables to sets of entities, while I map them to single entities. Labels are interpreted similarly, but in one case the resulting functions are applied to sets, while in the other they are applied to individuals.

1.4. Answers

An O-logic formula that is existentially quantified can be treated as a query for answers as well as a query for just a yes-no response. Consider a formula

$$\exists \texttt{X} \ \exists \texttt{Y} \ \exists \texttt{Z} \ f$$

This formula is true in a structure ST if there is an instantiation I with $M_{ST,I}(f) =$ true. I might be interested in the particular values of **X**, **Y** and **Z** that make the formula true. I define the *answers* to f under ST as

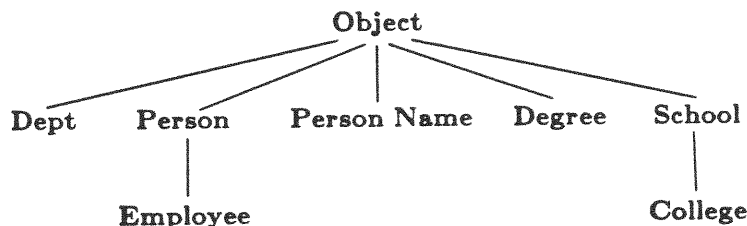
$\{I[\texttt{X Y Z}] \mid M_{ST,I}(f) = \text{true}\}$.

Here $I[\texttt{X Y Z}]$ is the restriction of the instantiation I to just a three-element domain of the variables of interest. $I[\texttt{X Y Z}]$ is essentially an **XYZ**-tuple, but it can have entities in it as well

as data values. Thus answers can actually reference pieces of the database.

2. A Class Hierarchy

To put some structure on the space of objects I introduce a class hierarchy. Assume a hierarchy H of classes, which need not be a strict tree.



I allow multiple inheritance in the hierarchy. The hierarchy is the first step towards a scheme for O-structure databases. Perhaps a fixed hierarchy of classes such as `String` and `Number` also exist to subdivide D. I modify O-terms to have a class name as a prefix.

Example 7: Redoing some O-terms from Example 2:

```
Employee: E

String: "Joe"

Employee:E(name ⟶ PersonName:N(first ⟶ String:"Joe", last ⟶ String:"Doe"),
           degree ⟶ Degree:D(level ⟶ String:"PhD", year ⟶ Number:1981,
             school ⟶ College:S(name ⟶ String:"SMU", state ⟶ String:"TX")))
```

I can omit classes in front of data values, because the class is implicit in the literal representation.

```
Employee:E(name ⟶ PersonName:N(first ⟶ "Joe", last ⟶ "Doe"),
           degree ⟶ Degree:D(level ⟶ "PhD", year ⟶ 1981,
               school ⟶ College:S(name ⟶ "SMU", state ⟶ "TX")))
```

2.1. Extending Structures to Hierarchies

I will augment structures to interpret classes in the hierarchy, and will modify the meaning function to check class prefixes on variables. A structure for an O-logic formula f is now a

triple $ST = <U, g, h>$, where U and g are as before. The function h maps the classes to sets of entities. (I assume that classes for data values are mapped in a fixed way.) So $h{:}H \longrightarrow 2^W$. The function h must respect the hierarchy. If **C1** is a superclass of **C2**, then $h(\texttt{C1}) \supseteq h(\texttt{C2})$. Note that an entity can have multiple, incomparable classes.

Now the meaning function checks for one more thing. Consider an O-term of the form **Cls:V** or **Cls:V(...)**. For $M_{ST,I}$ to be true of the term, $I(\texttt{V})$ must be in $h(\texttt{Cls})$.

Example 8: Consider a structure $ST = <U, g, h>$ where U and g are as before.

$U = \{e_1, e_2, e_3, e_4, e_5\} \cup D$

$g(\texttt{manager})(e_1) = e_2$, $g(\texttt{manager})(e_3) = e_4$, $g(\texttt{worksIn})(e_2) = e_1$, $g(\texttt{worksIn})(e_4) = e_5$,

$g(\texttt{dname})(e_1) =$ **"Sales"**, $g(\texttt{dname})(e_3) =$ **"Manuf"**, $g(\texttt{dname})(e_5) =$ **"RandD"**.

Let h be such that

$h(\texttt{Object}) = \{e_1, e_2, e_3, e_4, e_5\}$
$h(\texttt{Person}) = h(\texttt{Employee}) = \{e_2, e_4\}$
$h(\texttt{Dept}) = \{e_1, e_3, e_5\}$

And suppose I have the term

Dept:D(manager ⟶ Employee:M)

If I have an instantiation I such that $I(\texttt{D}) = e_1$ and $I(\texttt{M}) = e_3$, there is a problem right away, for although $I(\texttt{D}) \in h(\texttt{Dept})$, $I(\texttt{M}) \notin h(\texttt{Employee})$.

3. Storing Structures as a Database

My intent is that O-Logic be the basis of an object-oriented database. Initially, the idea is that an O-logic database will store a structure rather than a set of formulas. The structures will be entity finite. The database system will manage unique surrogates to identify entities in W. The g and h parts of the structure can be stored in various ways, such as binary and unary relations, or lists of field-value pairs. O-logic formulas can be evaluated against the structure. Existential formulas can be used to generate answers, which are tuples of entities and data

values. Later I will describe an intensional component to the database, and will attempt to define the semantics as the minimal model that satisfies the intensional formulas and extends the stored structure.

There isn't much in the way of a database scheme so far, apart from the class hierarchy. I could easily add typing constraints that say every member of a class has certain fields, and the values of those fields belong to certain classes.

Example 9: The formula

```
∀ E ∃ D ∃ E
      Employee:E  ⇒  E(worksIn → Dept:D,
                        name → PersonName:N)
```

says that every **Employee** entity has a **worksIn** field whose value is a **Dept** object and whose **name** is a **PersonName** object. Implications in this form could obviously be massaged into something that looks more like a scheme declaration:

```
Employee(worksIn → Dept, name → PersonName)
```

Note that such a declaration is *prescriptive*, not *proscriptive*. It says certain fields must exist, but an object may have more than the required fields. (Such latitude allows an object to belong to incomparable classes, such as **Employee** and **Stockholder**.) Certain restrictions on schemes between subclasses and superclasses must exist to admit any satisfying structure: the subclass must have at least the fields of the superclass, and the typing of those fields must be at least as restrictive as in the superclass. Schemes have much the same meaning as Ait-Kaci's ψ-terms [A-K]. However, here I have a data level as well as a type level. Everything in his system is a type.

Relations, if desired, are represented as tuple objects. A supplier-parts relation might have the scheme

```
SP(supplier → Company, provides → Part)
```

Such relations are non-1NF, because attribute values can be entities.

4. Updates

I now define updates on O-structures. Updates will be used to add existing entities to classes, set field values, and create new entities. The form of an *update command* is

$$t \Leftarrow f$$

where t is an O-term and f an O-logic formula. Formula f is used to provide a sequence of variable bindings, which will be used to instantiate t. Each instantiation of t is interpreted as a property that must be made true in the structure.

Example 10: I want to promote Joe Doe to manager of the Purchasing department.

```
Manager:E(manages ⟶ D) ⇐
      Employee:E(name ⟶ (first ⟶ "Joe", last ⟶ "Doe")),
      Dept:D(dname ⟶ "Purchasing")
```

For the moment, assume that f is an open formula that contains all the variables of t. Then the meaning of such an update command is that for all answers to f, modify the current structure so that t is true under all substitutions given by those answers. Any answer A for f can essentially be treated as an instantiation for t. For each such answer, I want to modify the stored structure ST so that $M_{ST,I}(t)$ = true.

Example 11: Continuing the last example, suppose there is just one answer A, where $A(\mathbf{E}) = e_1$ and $A(\mathbf{D}) = e_2$. Then I want to modify ST, if necessary, so that $E_1 \in h(\mathbf{Manager})$ and $g(\mathbf{manages})(e_1) = e_2$. Here I "create" the **manages** field for e_1 if it does not exist already. I will cover removing fields later.

If there were another answer B with $B(\mathbf{E}) = e_3$ and $B(\mathbf{D}) = e_2$ (two employees named "Joe Doe"), then I would also have to change ST so that $e_3 \in h(\mathbf{Manager})$ and $g(\mathbf{manages})(e_3) = e_2$.

Some updates will be disallowed because they overconstrain the structure. If answer B had been $B(\mathbf{E}) = e_1$ and $B(\mathbf{D}) = e_3$ (two Purchasing departments), then I must have both $g(\mathbf{manages})(e_1) = e_2$ and $g(\mathbf{manages})(e_1) = e_3$. In a later section, I show an extension of O-

structures to allow multivalued fields.

If f in $t \Leftarrow f$ contains all the variables of t, then the command can express only updates on class membership and field values. Now suppose in $t \Leftarrow f$ that t has some variables not occurring in f. I will treat an uninstantiated variable in the head as a request to create new entities. I do not want to satisfy the head of the update by conscripting some **Car** entity, say, to also play the role of **Manager**.

Example 12: I have a similar update command to the last one, but I assume Joe is being brought in from outside the company, and there is no entity in the database for him already.

```
Manager:E(manages ⟶ D, worksIn ⟶ D,
          name ⟶ PersonName:N(first ⟶ "Joe", last ⟶ "Doe"),
          age ⟶ 26) ⇐
 Dept:D(dname ⟶ "Purchasing")
```

In this case, for each answer A to f, I extend the answer by new entities not in ST on variables of t not in f, before modifying ST to make $M_{ST,I}(t)$ true.

Example 13: If A is an answer to the last update command body with $A(\mathbf{D}) = e_1$, I extend A to have $A(\mathbf{E}) = e_2$ and $A(\mathbf{N}) = e_3$, where e_2 and e_3 are entities not already in ST. I then modify ST so that

$e_2 \in h(\mathbf{Manager})$
$e_3 \in h(\mathbf{PersonName})$

$g(\mathbf{manages})(e_2) = e_1$
$g(\mathbf{name})(e_2) = e_3$
$g(\mathbf{age})(e_2) = 26$

$g(\mathbf{first})(e_3) =$ "Joe"
$g(\mathbf{last})(e_3) =$ "Doe"

Note that e_2 and e_3 must also be added to other classes in the hierarchy to make the changes above legal.

In practice, right sides of updates will be conjunctions of terms, and non-head variables will all be universally quantified, as has been the case with the examples in this section. Manchanda and Warren [MW] use a similar syntax for update rules, however the elementary update

actions are embedded in right sides of rules.

4.1. Limited Negation

For removing an entity from a class, or a field from an entity, I allow limited forms of *structural* negation in the head term.

Example 14: Take the manager of Purchasing out of the `Stockholder` class.

```
¬Stockholder:M  ⇐  Dept:D(name → "Purchasing",
                           manager → M)
```

Example 15: Remove the `worksIn` field of Joe Doe.

```
E(¬worksIn)  ⇐  Employee:E(name → PersonName(first → "Joe",
                                              last → "Doe")
```

Of course, an existing field value can be changed by asserting that field to have another value.

5. O-Rules

I now consider adding rules to a database. I restrict myself to "Horn"-type rules, as I want to have a minimal model semantics for a database with rules. (By Horn, I mean that the right side is a conjunction of O-terms, and the head has no negations.) As it turns out, certain kinds of rules will require fudging the meaning of minimal a bit.

I will call Horn rules for O-logic *O-rules*. O-rules will have the same syntax as updates, except for `:-` in place of $\Leftarrow$.

The guiding principle for the semantics of O-rules is that they behave like deferred updates. It is as if I execute all the rules as updates just before answering a query. This interpretation is more or less the minimal model semantics of Prolog. However, for O-logic I have to be careful, because an update can change a field value. Thus, it is possible to have a rule whose head is an O-term that contradicts information in the database. Field notation in O-logic gives a limited kind of equality, which is enough to allow a contradiction to be expressed without negation, given a fixed interpretation of data values in D.

```
V(fld → 1) ∧ V(fld → 2)
```

While in general it is probably undecidable whether an O-rule conflicts with the database, there are some innocuous syntactic conditions that can handle the problem, such as designating each field as virtual or stored. An O-rule that adds an entity to a class is never a problem, as it cannot contradict stored information. Such a rule might be redundant, however.

An O-rule that has a variable in the head that is not in the body (an *entity-creating* rule) also cannot cause a contradiction, but poses a problem for defining a minimum model semantics. As a stab at a minimal model definition, I might say that the meaning of a structure $ST = <U, g, h>$ under a set of rules is the structure $ST' = <U', g', h'>$ such that ST' contains ST, the rules interpreted as updates do not add anything to ST', and no structure smaller than ST' has those two properties. Containment is interpreted componentwise:

$U' \supseteq U$,

g'(**lab**) is defined at least where g(**lab**) is, for every label **lab**,

h'(**C**) $\supseteq$ h(**C**) for every class **C**.

O-rules that add fields or add entities to classes are not a problem for minimality. I can imagine applying such rules as updates until no more changes occur. Entity-creating rules are another matter. Note that "applying the rule as an update" is critical. If an entity already exists that satisfies the head of such a rule, the rule adds another entity to ST' anyway. Also, it does not appropriate an existing entity and move it to a new class to satisfy the head. While that approach can give a smaller model, it would mean giving up any hope that ST' could be uniquely defined. For an entity-creating rule, since the corresponding update introduces a new entity, I could in principle apply the rule over and over, generating more and more entities. Thus, there would be no ST' that was unchanged under the rules, even allowing the set of entities in U' to be infinite.

The problem with entity-creating rules is the thorniest issue in O-logic, I believe, or in formal semantics of any system that models unique identity. There are similar problems with *new* as a storage allocation function in programming languages. Talking to other people trying to do formal object models, they have the same problem. Also, entity-creating rules are analogous to embedded tuple-generating dependencies in relational dependency theory, which is a particularly recalcitrant class to reason about.

I think the answer lies in a meta-axiom that limits application of entity-creating rules. Recall that an update that creates a new entity takes an answer A for the body of the command and extends it to the variables in the head of the command. The meta-axiom says that any one answer gets extended exactly once. Returning to the update that adds Joe Doe as a manager, and viewing it as a rule, the meta-axiom says that exactly one set of bindings of **E** and **N** are made for each binding of **D**.

```
Manager:E(manages ⟶ D, worksIn ⟶ D,
          name ⟶ PersonName:N(first ⟶ "Joe", last ⟶ "Doe"),
          age ⟶ 26) :-
 Dept:D(dname ⟶ "Purchasing")
```

This axiom, I believe, allows me to say when a rule has been applied "enough" times, and so gives a well-defined notion of minimal model. In the next section I describe a way to implement entity-creating rules under this meta-axiom.

5.1. Skolem Surrogates

I now look more closely at quantification in entity-creating rules. Consider an entity-creating rule that forms an **InterestingPair** entity for each employee whose manager has the same last name. I define the following rule.

```
InterestingPair:P(emp ⟶ E, man ⟶ M) :-
   Employee:E(name ⟶ PersonName:(last ⟶ LN),
              worksIn ⟶ Dept:(manager ⟶ Employee:M(
                                      name ⟶ PersonName:(last ⟶ LN)))).
```

The first problem is the quantification of object variable **P** in the head O-term. $\forall$ **P** does not

make sense here, since I do not want to say that every object (or even every **InterestingPair** object) has **E** in its **emp** field and **M** in its **man** field. What I really want is a "there exists a distinct" quantifier to match the update semantics. That is, there is some **InterestingPair** object **P** for each **E** and **M** value that the body matches.

One approach is to "Skolemize" the new objects a rule posits. In the **InterestingPair** rule above, I see the implicit quantification as

$$\forall \mathbf{E} \forall \mathbf{M} \forall \mathbf{N} \exists \mathbf{P}.$$

(Since the quantifiers can be different, their order matters.) I assume one new **InterestingPair** object for each <**E**, **M**, **N**> triple that matches the body of the rule. For $<e_4\ e_5\ e_{19}>$ say, I could create a "Skolem surrogate" **ip**$[e_4, e_5, e_{19}]$ for the new **InterestingPair** object. With this approach, whenever I need an object implied by a rule in matching a goal, I will create an object with a Skolem surrogate in temporary object space, being careful not to create an object with the same surrogate twice. The version of minimality I get with this approach to rules is consonant with the situation in Prolog where entities with different names are considered different. That is, assume entities are different in the absence of other information. This approach agrees with the meta-axiom of extending each answer once in applying an entity-creating O-rule. The constant **ip**$[e_4, e_5, e_{19}]$ is the unique extension of the answer $<e_4\ e_5\ e_{19}>$ for the body of the rule.

5.2. Top-Down Evaluation of Rules

Closing up the O-structure stored in the database under O-rules is not an attractive way to process queries against a database with an intensional component. I would like a top-down mechanism to chain backwards from a query, via the rules, to the stored structure. Given an O-term, I want to solve it by unifying it with the head of an O-rule or matching it against an entity in the structure. There is a twist here, in that there may be a "remainder" upon unification. Consider for the moment that I have government employees, whose pay is based on their GS level, and government managers as a subclass. Say

```
GovEmp(level → Grade, pay → Money)
GovMan(level → Grade, pay → Money, manages → Dept)
```

(I have omitted other fields for simplicity) with the rule

```
E(pay → P) :-
     GovEmp:E(level → L) ∧ rate(level → L, pay → P).
```

Now, if I have a goal

```
GovMan:M(pay → 28000, manages → Dept:D(dname → "Forecasting"))
```

I can use the rule to try to match the **pay → 28000**, generating the subgoals

```
GovEmp:M(level → L), rate(level → L, pay → 28000)
```

However, I am still left with a remainder of

```
GovMan:M(manages → Dept:D(dname → "Forecasting"))
```

from the original goal. The rule only takes care of the **28000** part if the subgoals succeed. I still have to show that **M** is a **GovMan** and that the department for **M** is **"Forecasting"**.

The existence of a remainder upon resolution is not surprising if I look at a translation of fields to unary and binary relations in regular logic. For the goal I get a translation

```
pay(M, 28000) ∧ manages(M, D) ∧ dname(D, "Forecasting")
        ∧ govMan(M) ∧ dept(D)
```

while the head of the rule translates to just

```
pay(E, P) :- ...
```

which solves only one part of the goal. The translation of the head into regular logic shows that "Horn" O-logic is really something more than conventional logic. The head of the **InterestingPair** rule translates to

```
emp(P, E) ∧ man(P, M) ∧ interestingPair(P)
```

which cannot be translated into two Horn rules, because of the shared variable. In a database system that has rules much like O-rules, we were able to translate all rules into function-free

Prolog, except entity-creating rules [Zh]. Those rules required a new interpreter that created temporary objects when applying entity creating rules. Creating temporary objects allowed us to handle rules with a conjunction of literals in the head, as well as avoiding using the same rule twice with the same binding. An alternative translation, suggested by Catriel Beeri and Oded Shmueli, uses function symbols for Skolem surrogates, and dispenses with temporary objects.

Note that ψ-term unification as defined by Ait-Kaci [A-K] does not do the right thing here (although it can be used in the government employee example to infer type information about `M`). The rule matches the goal under ψ-term unification, binding `E` to `M` and `P` to `28000`, leaving the goals

```
GovEmp:M(level ⟶ L), rate(level ⟶ L, pay ⟶ 28000)
```

It thereby loses the requirements on `M` being a manager and in the Forecasting department. This is no defect with ψ-term unification per se, the inference step above makes sense for statements about types. It just does not give a valid inference rule with my semantics for terms.

Pereira and Shieber [PS] have given a meaning to grammar productions that include feature structures. The denotational semantics they give for the feature part of such rules may help out here, although their semantics is expressed as bottom-up application of rules, and I have yet to apply it for top-down processing. Mukai [Mu] has proposed a style of unification, for terms resembling O-terms, that may give rise to a resolution procedure for top-down evaluation of O-rules.

6. Extensions

6.1. Rules as Objects

The meta-predicates that give Prolog the reflexivity to treat programs as data have obvious utility for code generation and translation applications. Letting O-rules be entities in the stored structure has other interesting implications. Such a rule could contain a reference to an

actual object in place of an O-term. Rather than having to describe a particular department unambiguously, I can construct a rule that references the object for that department directly. Referencing an object in the head of a rule allows the rule to be specialized to that object alone.

6.2. Negation

I showed limited negation in the heads of updates. While I do not want to support negation in the head of a rule, I see no problem with structural negation in terms in the body of a rule.

Example 16: The term

```
¬Manager:E
```

matches any entity not in the **Manager** class. The term

```
Dept:D(¬budget)
```

matches a **Dept** object with no **budget** field, while

```
Dept:D(budget → ¬1140000)
```

matches a **Dept** object with a budget other than $1,140,000.

6.3. Multi-fields

I do not see insurmountable problems with allowing multiple occurrences of a field in a structure, such as a department being assigned to more than one building. An O-term

```
Dept:D(building → B)
```

as a query would have multiple answers with the same department entity if that department were in more than one building. Multi-fields also mean some kinds of transitive closure rules can be expressed without creating new objects.

```
E(superior → M) :- E(worksIn → (manager → M)).
E(superior → M) :-
      E(worksIn → (manager → N)) ∧ N(superior → M).
```

A set can be modeled as an object with a single multi-field:

```
EmpSet(element → Employee)
```

Why not introduce sets as a primitive concept rather than multi-fields? I think it will be useful for a database designer to decide whether or not to confer "objecthood" on a group of values that represent a property of some entity. Multifields support sets, while sets do not support multi-fields.

Some places for care are update and negation. Field updates to multi-fields must distinguish between adding a new occurrence of the field and changing the value of an existing occurrence. With negation, there is the subtle distinction between "none of the buildings for the department is C50" and "some building for the department is not C50."

7. Conclusion

I have given a logic based on terms that represent the existence and internal structure of objects, rather than names of objects and relationships among those names. The logic incorporates a hierarchy to arganize classes of objects, and structures in this logic are amenble to database representation. Updates in the lpogic preserve identity of entities, and provide the foundation for the semnatics of rules, which are treated as deferred updates. I pointed out some challenges facing the top-down evaluation of rules, which will be the next area for conintued research.

8. Acknowledgements

Thanks to Harry Porter, Carlo Zaniolo, Dick Tsur, Catriel Beeri and Oded Shmueli for discussions of this material. This work was supported by NSF grant IST 83 51730 (cosponsored by Tektronix Foundation, Intel, Digital Equipment, Servio Logic, Mentor Graphics, Xerox, Beaverton Area Chamber of Commerce, IBM) and contracts from Tektronix Computer Research Laboratory and the MCC Database Program.

9. Bibliography

[A-K] Hassan Ait-Kaci, *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures,"* PhD Thesis, Univ. of Pennsylvania, 1984.

[A-KN] Hassan Ait-Kaci and Roger Nasr, *LOGIN: A logic programming language with built-in inheritance,* Journal of Logic Programming 3 (3), October 1986.

[Ca] R. G. G. Cattell, *Design and implementation of a Relationship-Entity-Datum model,* Xerox CSL 83-4, May 1983.

[GM] Joseph A. Goguen and Jose Meseguer, *Extensions and foundations of object-oriented programming,* presented at the Workshop on Object-Oriented Programming, IBM T. J. Watson Research Center, June 1986, SIGPLAN Notices 21 (10), October 1986.

[KC] S. N. Khoshafian and G. P. Copeland, *Object identity,* Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, September-October 1986.

[Ko] R. Kowalski, *Database updates in the event calculus,* presented at the Workshop on Foundations of Deductive Databases and Logic Programming, August 1986.

[Ma] B. J. MacLennan, *A view of object oriented programming,* Naval Postgraduate School NPS52-83-001, February 1983.

[MW] S. Manchanda and D. S. Warren, *Towards a logical theory of database view updates,* presented at the Workshop on Foundations of Deductive Databases and Logic Programming, August 1986.

[MG] D. S. Moffat and P. M. D. Gray, *Interfacing Prolog to a persistent data store,* Proceedings for the 3rd Intl. Conf. on Logic Programming, July 1986.

[Mo] Richard Montague, *The proper treatment of quantification in ordinary English,* in *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics,* Reidel Publ. Co., 1973.

[Mu] K. Mukai, *Anadic tuples in Prolog,* presented at the Workshop on Foundations of Deductive Databases and Logic Programming, August 1986.

[PS] Fernando C. N. Pereira and Stuart M. Shieber, *The semantics of grammar formalisms seen as computer languages,* 10th Intl. Conf. on Computational Linguistics, July 1984{