C-Logic of Complex Objects

Weidong Chen and David S. Warren
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract

Our objective is to have a logical framework for natural representation and manipulation of complex objects. We start with an analysis of semantic modeling of complex objects, and attempt to understand what are the fundamental aspects which need to be captured. A logic, called C-logic, is then presented which provides direct support for what we believe to be basic features of complex objects, including object identity, multi-valued labels and a dynamic notion of types. C-logic has a simple first-order semantics, but it also allows natural specification of complex objects and gives us a framework for exploring efficient logic deduction over complex objects.

1 Introduction

Reasoning about complex objects has been the focus of active research in databases during the last few years [1,2,6,7] [8,12,13,14,20,22,23]. One feature common to most of these proposals is that they are language-based. That is, a language is first presented for describing complex objects, and the semantics of the language is then given. This language-based approach has two disadvantages. First, different languages have various features, and it is not always clear why they are needed and how they support complex objects. Even the same feature may have subtle semantic differences in different languages. This makes it difficult to compare the semantics of complex objects across different languages. Second, each language may impose certain seemingly unnecessary constraints over complex object specification, which can make the language inflexible.

The primary purpose of complex objects is to capture more of the structure of real world data. Our strategy is to focus on complex object modeling instead of specific languages. And the design of a language should be guided by what need to be modeled, rather than the other way around. This paper consists of two major parts. First, we start with an analysis of semantic modeling of complex objects, and try to identify fundamental aspects of complex objects which need to be captured. An informal discussion is given about identities, labels and types of complex objects, and how they might be modeled in a logical framework. Second, based upon this analysis, we present a logic, called C-logic, to support what we believe to be essential features of complex objects, including object identity, multi-valued labels and a dynamic notion of types. A transformation into first-order logic is then described, which provides an alternative way of understanding complex objects in a well-known framework.

The design of C-logic has been stimulated by Maier's O-logic [22]. O-logic contains notions such as object identity, labels and types, which are essential for modeling complex objects. As Maier pointed out in [22], however, O-logic also has some problems. C-logic can be viewed as an extension of O-logic in the sense that it attempts to clarify and solve some problems in O-logic.

In addition to the semantics of complex objects, simplicity and flexibility are also guiding principles for us in the design of C-logic. First, C-logic is not just yet another logic for complex objects. Its semantics is first-order and can be understood easily. Each formula in C-logic can actually be transformed into an equivalent first-order formula. It directly supports only what we view as fundamental aspects of complex objects. Other higher-level features such as single-valued labels and a static notion of types can be added on top of C-logic. This makes C-logic very simple and easily implementable. Second, C-logic is much more than just first-order logic. It supports many useful aspects of sets. And it enables the user to provide clustering information in the specification of complex objects. More efficient logic deduction can be done directly over complex objects by taking advantage of this information. The simplicity of C-logic makes it a suitable paradigm for such exploration.

2 Complex Object Modeling

The objective of complex objects is to model structured entities in the real world. Each entity can have various properties and can be classified into various classes according to these properties. In order to model real world entities, we need corresponding representations of entities, their properties, and classes, using identities, labels and types. In the following, we discuss informally how each of the three aspects will be treated in C-logic.

2.1 Object Identity

In the real world, the actual existence of an entity in time and space identifies the entity. To model such a situation directly, we need to represent the identity of an entity. The importance of object identity has been argued in [17,21,22]. The question is how to represent identities in a logic framework.

In Maier's O-logic [22], only object variables are introduced for referring to object identities. This is insufficient, as Maier points out, when objects are defined by rules. Consider the following example:

```
path: C(src ⇒X, dest ⇒Y, length ⇒1) :-
      node: X(linkto ⇒Y).
path: C(src ⇒X, dest ⇒Y, length ⇒L) :-
      node: X(linkto ⇒Z),
      path: C0(src ⇒Z, dest ⇒Y, length ⇒L0),
      L is L0 + 1.
```

In O-logic, such rules are considered as entity-creating rules. However, these rules themselves do not determine what entities are to be created, i.e., how C should be quantified with respect to other variables in the rules. It was recognized by Maier that C should be existential. But the scope of the existential quantifier remains unspecified. There are several possible cases, each of which is reasonable and each of which has a different semantics. The path objects can be determined by either

1. node objects at both ends only; or
2. node objects at both ends and the length of the path only; or
3. the sequence of node objects of the path.

Consider variables in the second rule. The quantification would be $\forall X \forall Y \exists C \cdots$ for the first case, and $\forall X \forall Y \forall L \exists C \cdots$ for the second. For the third case, one possible quantification of variables is $\forall X \forall C0 \exists C \cdots$. The user should specify which is the intended semantics. The skolem function of the existential variable could then be automatically generated or explicitly given by the user. For example, if path objects are determined by node objects at both ends only, the rules would become

```
path: id(X,Y)[src ⇒X, dest ⇒Y, length ⇒1] :-
        node: X[linkto ⇒Y].
path: id(X,Y)[src ⇒X, dest ⇒Y, length ⇒L] :-
        node: X[linkto ⇒Z],
        path: C0[src ⇒Z, dest ⇒Y, length ⇒L0],
        L is L0 + 1.
```

Notice that the existential object variable C in each rule has been replaced by id(X,Y). As we have said, this replacement can be done by either the system or the user.

A logic of complex objects should allow explicit construction of object identities. As proposed in [18], we provide constants and functions for constructing object identities in C-logic. Our original motivation for introducing structured object identities is to support skolemization of existential object variables. However, this does not mean that the user need be worried about constructing unique identities. Instead, based upon C-logic, a high-level interface can be built, in which the user specifies only what determines the objects to be created, but not how to construct identities of these objects. The acutal construction of unique identities can be left to the system supporting C-logic. Given such a high-level interface, the user, in the above example, would not give the explicit construction id(X, Y) of identities, but only that object variable C in the original rules is existentially dependent upon X and Y.

2.2 Labels as Properties

Properties of an object can be represented by labeled values. For example, consider the fact that the *author* of the book *Foundations of Logic Programming* is *John W. Lloyd*. In other words, the property *author* of a book object *Foundations of Logic Programming* has value *John W. Lloyd*. Labels have been introduced in various languages of complex objects. However, it is not always clear what these labels mean [3,4,6,12].

In O-logic [22], labels are considered semantically as partial functions from objects to objects. A program containing a multiply-defined label would have no models. So even if a program contains only Horn-like rules, it may still be inconsistent. Consistency checking of a program essentially requires evaluating the whole program, which may be difficult if rules are involved, even undecidable if there are structures.

While inconsistency in O-logic is global, there are some other proposals [6,18] which employ a lattice-based semantics of objects and introduce a top object $\top$. The idea is that if a label is multiply defined, $\top$ would be derived for the value of the label. It has been pointed out in [18] that introducing $\top$ makes inconsistency local to objects and labels being concerned. For example, suppose that

```
john[name ⇒"John"].
john[name ⇒"John Smith"].
```

and "John" and "John Smith" do not have any common super-object except $\top$. Then in the semantics of the data base, john[name ⇒$\top$] would be true. Therefore john[name ⇒"David"] should also be derivable since it is a sub-object of john[name ⇒$\top$]. However, this derivation cannot be

done using only resolution-like inference rules.

In C-logic, we consider labels as binary predicates over objects. There are many cases in which such multi-valued labels are very convenient. For example, a person may have several telephone numbers and several children. A student may have several co-advisors. Multi-valued labels do not have the builtin functionality constraint, and thus are easier to implement. For a logic supporting basic features of complex objects, we do not want to build constraints into the logic which are hard to enforce.

Viewing labels as binary predicates has another implication. Since each label is a binary predicate, a description of an object with several labeled values can be viewed as a conjunction of several atomic formulas. For example,

john[name ⇒"John Smith", age ⇒28]

can be considered as

john[name ⇒"John Smith"] ∧ john[age ⇒28]

or as

name(john, "John Smith") ∧ age(john, 28)

in first-order logic. There are three aspects worth pointing out. First, from a complex description of an object, we can infer any sub-part of the description of the object. So john[name ⇒"John Smith"] and john[age ⇒28] can be infered from john[name ⇒"John Smith", age ⇒28]. Second, we can combine various pieces of descriptions together to infer a complex one. For instance, given john[name ⇒ "John Smith"] and john[age ⇒28], we can infer john[name ⇒ "John Smith", age ⇒28]. This is important since information about an object may be accumulated piecewise. Third, the translation of complex object descriptions into first-order logic provides an understanding of complex objects within a well-known logic system, as well as a simple implementation. One difference, however, between C-logic and first-order logic is that C-logic encourages structured descriptions of complex objects. While a complex description is very natural in C-logic, programming using conjunctions of atomic formulas in first-order logic is not so common.

2.3 Types of Objects

Thus far we have argued that objects must have both identities and properties or labeled values. Objects can be divided into various classes according to properties they have. In databases, a class of objects has two aspects. Under the dynamic aspect, a class denotes the set of objects (or object identities) in the class, and such membership may be changed by database updates. The static aspect represents common structural properties of all objects in the class, i.e., what properties an object must have in order to belong to a certain class. This may also be changed if the database schema is modified. Corresponding to these two aspects, there are two different notions of types: dynamic and static.

In a dynamic notion of types, a type is simply a set of object identities. It is essentially part of the database state. There are no additional assumptions regarding what properties an object must have in order to be in a certain type. Instead, whenever an object is added to the database, it must also be specified what type the object will be in. (A default type, such as *object*, which includes all objects, may also be used.) Each type can be treated as a unary predicate.

In a static notion of types, the meaning of a type is not so clear. Roughly speaking, a type indicates a set of properties which must be possessed by objects of that type. Logically, let $l_1, \cdots, l_n$ be labels corresponding to all properties indicated by a type τ. Then one possible meaning of τ is a set of objects specified as follows:

$$\tau(X) :- X[l_1 \Rightarrow X_1, \cdots, l_n \Rightarrow X_n].$$

Of course, all objects $X_1, \cdots, X_n$ can be further typed. The point is that every object with all properties specified by a type will automatically belong to the type.

Types can be organized into hierarchies. In a dynamic notion of types, the type hierarchy has to be explicitly specified, while in a static notion of types, the hierarchy is implicitly determined by properties of each type. In C-logic, we choose to use a dynamic notion of types. The reason is that the static notion is a kind of constraint which seems better treated with schema information and other constraints over the database state such as functionality of labels. Since we will not deal with constraints, we use the dynamic notion of types to make the framework simpler.

3 Basic Framework

This section describes the formal syntax and semantics of C-logic, and presents a transformation of this logic into first-order logic. The transformation provides an alternative way of understanding complex objects, and a basis for reasoning about complex objects in first-order logic. It also indirectly establishes (by the Herbrand theorem of first-order logic) that mechanical reasoning about complex objects corresponds to complete pure logic deduction.

3.1 Languages of Objects

The basic syntax is as follows. In addition to parentheses and logical connectives ($\wedge$ $\vee$ $\neg$ $\supset$ $\forall$ $\exists$), a language of objects contains:

- a countably infinite set of variables;
- a (countable, possibly empty) set of (possibly zero-ary) function symbols;
- a (countable, possibly empty) set of predicate symbols;
- a (countable, possibly empty) set of labels;
- a countable, partially ordered set of type symbols,

which contains a type symbol *object* such that for any type ι, $\iota \leq object$.

Assume that all of these sets of symbols are disjoint.

Let ι be any type symbol. A term is either

- $\iota : X$ where X is a variable; or
- $\iota : c$ where c is a constant; or
- $\iota : f(t_1, \cdots, t_n)$, where f is an n-ary function symbol, t_i $(1 \leq i \leq n)$ is a term; or
- $t[l_1 \Rightarrow e_1, \cdots, l_n \Rightarrow e_n]$ $(n \geq 1)$, where t is a term of the form $\iota : X$, $\iota : c$, or $\iota : f(s_1, \cdots, s_k)$, and l_i $(1 \leq i \leq n)$ is a label, and e_i $(1 \leq i \leq n)$ either a term or a collection of terms of the form $\{t_1^i, \cdots, t_{n_i}^i\}$ in which $t_1^i, \cdots, t_{n_i}^i$ are all terms.

As a notational convenience, a term of the form $object : t$ can be abbreviated as t.

Example 1: The following

```
X
path: g(X,Y)[length ⇒10]
person: john[children ⇒
                    {person: bob, person: bill}]
instructor: david[course ⇒courseid: cse538,
                    course ⇒courseid: cse505]
```

are terms, but

```
student: id[name ⇒joe][age⇒20]
part: f(part_id ⇒123)
part: f[part_id ⇒123]
```

where f is a unary function symbol, are not terms.

Intuitively, a term like $\iota : t[l_1 \Rightarrow t_1, \cdots, l_n \Rightarrow t_n]$ represents an object of type ι, whose identity is t, with certain properties indicated by $l_i \Rightarrow t_i$ $(1 \leq i \leq n)$.

An atomic formula is either $p(t_1, \cdots, t_n)$ or t, where p is an n-ary predicate symbol, and t, t_i $(1 \leq i \leq n)$ are terms. Formulas can be constructed from atomic formulas using logical connectives in the usual way.

3.2 Semantics of Languages of Objects

Given a language **L** of objects, a semantic structure is a pair $\mathbf{M} = (M, I)$ where M is a nonempty set called the domain of the structure, and I is the interpretation function which assigns values to nonlogical symbols in **L** as follows ($\mathcal{P}$ is the powerset operator):

- $I(f) \in [M^n \rightarrow M]$ for every n-ary function symbol f;
- $I(p) \in \mathcal{P}(M^n)$ for every n-ary predicate symbol p;
- $I(l) \in \mathcal{P}(M^2)$ for every label l;
- $I(\iota) \in \mathcal{P}(M)$ for every type ι.

such that for any two types ι_1 and ι_2, if $\iota_1 \leq \iota_2$, then $I(\iota_1) \subseteq I(\iota_2)$. Notice that a label is semantically the same as a binary predicate and so is possibly multi-valued (or non-functional). And a type is semantically the same as a unary predicate. However, labels and types are pragmatically different from predicates in a language of objects. They can appear inside terms while predicates cannot. For convenience, we will also use $a_{\mathbf{M}}$ to denote $I(a)$ for each nonlogical symbol a.

Since formulas and terms may contain free variables, their semantics usually depends upon specific variable assignments. A term will have two meanings since it can be used both for denoting an object and for indicating whether the denoted object satisfies certain properties. Given a language **L** of objects, let

α be a formula in **L**,
M a semantic structure for **L**,
$s : [V \rightarrow M]$ a function from the set V of all variables into the domain M of **M**.

Let *Term* denote the set of all terms in **L**. We recursively define the extension $\bar{s}_{\mathbf{M}} : Term \rightarrow M$ of a variable assignment s to the set of all terms as follows:

- For each variable X, $\bar{s}_{\mathbf{M}}(\iota : X) = s(X)$.
- For each constant c, $\bar{s}_{\mathbf{M}}(\iota : c) = c_{\mathbf{M}}$.
- If $t_1, \cdots, t_n$ are terms and f is an n-ary function symbol, then

$$\bar{s}_{\mathbf{M}}(\iota : f(t_1, \cdots, t_n)) = f_{\mathbf{M}}(\bar{s}_{\mathbf{M}}(t_1), \cdots, \bar{s}_{\mathbf{M}}(t_n)).$$

- If t is a term of the form ι: X, ι : c, or $\iota : f(t_1, \cdots, t_m)$, and l_i $(1 \leq i \leq n)$ is a label, then

$$\bar{s}_{\mathbf{M}}(t[l_1 \Rightarrow e_1, \cdots, l_n \Rightarrow e_n]) = \bar{s}_{\mathbf{M}}(t)$$

where e_i $(1 \leq i \leq n)$ either a term or a collection $\{t_1^i, \cdots, t_{n_i}^i\}$ of terms.

We now define what it means for **M** to *satisfy* α *with* s,

$$\mathbf{M} \models \alpha[s]$$

as follows:

- For every variable X, $\mathbf{M} \models \iota : X[s]$ if $s(X) \in \iota_{\mathbf{M}}$;
- For every constant c, $\mathbf{M} \models \iota : c[s]$ if $c_{\mathbf{M}} \in \iota_{\mathbf{M}}$;
- If $t_1, \cdots, t_n$ are terms and f is an n-ary function symbol, then $\mathbf{M} \models \iota : f(t_1, \cdots, t_n)[s]$ if $\bar{s}_{\mathbf{M}}(\iota : f(t_1, \cdots, t_n)) \in \iota_{\mathbf{M}}$ and $\mathbf{M} \models t_i[s]$ for every $i(1 \leq i \leq n)$.
- If t is a term of the form ι: X, ι : c, or $\iota : f(t_1, \cdots, t_m)$, and l_i $(1 \leq i \leq n)$ is a label, then $\mathbf{M} \models t[l_1 \Rightarrow e_1, \cdots, l_n \Rightarrow e_n][s]$ if $\mathbf{M} \models t[s]$, and for every e_i, either
  - e_i is a term, $\mathbf{M} \models e_i[s]$ and $< \bar{s}_{\mathbf{M}}(t), \bar{s}_{\mathbf{M}}(e_i) > \in (l_i)_{\mathbf{M}}$: or
  - e_i is a collection of terms of the form $\{t_1^i, \cdots, t_{n_i}^i\}$, and for every $j(1 \leq j \leq n_i)$, $\mathbf{M} \models t_j^i[s]$

and $< \bar{s}_{\mathbf{M}}(t), \bar{s}_{\mathbf{M}}(t'_j) > \in (l_i)_{\mathbf{M}}$.

- If $t_1, \cdots, t_n$ are terms and p is an n-ary predicate symbol, then $\mathbf{M} \models p(t_1, \cdots, t_n)[s]$ if $\mathbf{M} \models t_i[s]$ for every $i (1 \leq i \leq n)$ and $< \bar{s}_{\mathbf{M}}(t_1), \cdots, \bar{s}_{\mathbf{M}}(t_n) > \in p_{\mathbf{M}}$.

The meaning of general formulas can be defined in the usual way from those of atomic formulas.

The semantics of C-logic has the following property. A term of the form $t[l_1 \Rightarrow t_1, \cdots, l_n \Rightarrow t_n]$ is semantically equivalent to $t[l_1 \Rightarrow t_1] \wedge \cdots \wedge t[l_n \Rightarrow t_n]$; a term of the form $t[l \Rightarrow \{t_1, \cdots, t_n\}]$ is semantically equivalent to $t[l \Rightarrow t_1] \wedge \cdots \wedge t[l \Rightarrow t_n]$. Thus a complex object description can always be decomposed into atomic descriptions involving only one label, and various pieces of descriptions can be combined into a complex one.

Another aspect is that C-logic has both terms and predicates for formulas and they are semantically different. For example, from the following two facts in which p is a constant denoting an object:

```
p[src ⇒a, dest ⇒b].
p[src ⇒c, dest ⇒d].
```

we can infer p[src ⇒a, dest ⇒d] or p[src ⇒c, dest ⇒b]. However, given

```
p(a, b). p(c, d).
```

in which p is a binary predicate, we cannot infer either p(a, d) or p(c, b). The difference is that labels of a term are independent, while arguments in a tuple of a predicate are associated together. This has certain implications with regard to query evaluation which will be discussed later.

3.3 Transformation into First-Order Logic

We show that, for any formula in a language of objects, there exists an equivalent genuine first-order logic formula. This transformation into first-order logic can be used to provide the basis for implementing complex object reasoning in first-order logic. Since formulas are freely generated from atomic formulas by logical connectives, we will show only how an atomic formula in a language of objects can be transformed into a conjunction of first-order atomic formulas. There are three levels involved in a transformation into first-order logic: language, semantic structure, and formula.

Given any language $\mathbf{L}$ of objects, there is a corresponding first-order language $\mathbf{L}^*$ whose alphabet contains the following sets of symbols:

- logical connectives: $\wedge \vee \neg \supset \forall \exists$;
- auxiliary symbols: "(", ")" and ",";
- the countably infinite set of variables of $\mathbf{L}$;
- the set of function symbols of $\mathbf{L}$;
- the set of predicate symbols of $\mathbf{L}$;
- a binary predicate symbol l for each label l in $\mathbf{L}$;
- a unary predicate symbol ι for each type ι in $\mathbf{L}$.

As a first-order logic language, $\mathbf{L}^*$ has its own definition of individual terms and first-order formulas as usual.

Semantically, for every structure $\mathbf{M}$ $(= (M, I))$ of $\mathbf{L}$, there is a first-order structure $\mathbf{M}^*$ $(= (M, I))$ of $\mathbf{L}^*$ which is essentially the same. However, a semantic structure for $\mathbf{L}^*$ is not necessarily a semantic structure for $\mathbf{L}$ because it may not respect the type hierarchy of $\mathbf{L}$. We define a set $TA_{\mathbf{L}^*}$ of first-order formulas called the type axioms of $\mathbf{L}^*$. For any two type symbols ι_1 and ι_2 such that $\iota_1 \leq \iota_2$, $TA_{\mathbf{L}^*}$ contains a formula $\forall X(\iota_1(X) \supset \iota_2(X))$, which is also written in Prolog's convention as

$$\iota_2(\mathrm{X}) :- \iota_1(\mathrm{X}).$$

Then any first-order semantic structure of $\mathbf{L}^*$ satisfying $TA_{\mathbf{L}^*}$ would also correspond to a semantic structure of $\mathbf{L}$. Obviously, the correspondence is one-to-one between semantic structures of $\mathbf{L}$ and semantic structures of $\mathbf{L}^*$ that satisfy $TA_{\mathbf{L}^*}$.

Theorem 1 *For any atomic formula α in a language $\mathbf{L}$ of objects, there exists a first-order formula α^* in $\mathbf{L}^*$ such that*

1. *For any semantic structure $\mathbf{M}$ of $\mathbf{L}$ and any variable assignment s, $\mathbf{M} \models \alpha[s]$ iff $\mathbf{M}^* \models \alpha^*[s]$;*
2. *For any semantic structure $\mathbf{M}^*$ of $\mathbf{L}^*$ satisfying $TA_{\mathbf{L}^*}$, there exists a semantic structure $\mathbf{M}$ of $\mathbf{L}$ such that for any variable assignment s, $\mathbf{M}^* \models \alpha^*[s]$ iff $\mathbf{M} \models \alpha[s]$.*

Proof: Since the correspondence is one-to-one between semantic structures of $\mathbf{L}$ and semantic structures of $\mathbf{L}^*$ that satisfy $TA_{\mathbf{L}^*}$, we will just prove (1).

Given an atomic formula in $\mathbf{L}$, we will construct an equivalent conjunction of first-order atomic formulas in $\mathbf{L}^*$. First, given a term t, we construct a first-order term t' as follows:

- $(\iota : X)' \equiv X$ for every variable X;
- $(\iota : c)' \equiv c$ for every zero-ary function c;
- $(\iota : f(t_1, \cdots, t_n))' \equiv f(t'_1, \cdots, t'_n)$;
- $(t[l_1 \Rightarrow t_1, \cdots, l_n \Rightarrow t_n])' \equiv t'$.

It can be proved by induction that, given any term t, $\bar{s}_{\mathbf{M}}(t) = \bar{s}_{\mathbf{M}^*}(t')$ for every semantic structure $\mathbf{M}$ of $\mathbf{L}$ and every variable assignment s.

Given an atomic formula α, we construct a conjunction α^* of first-order atomic formulas as follows:

- $(\iota : X)^* \equiv \iota(X)$ for every variable X;
- $(\iota : c)^* \equiv \iota(c)$ for every zero-ary function c;
- $(\iota : f(t_1, \cdots, t_n))^* \equiv$
  $\iota((\iota : f(t_1, \cdots, t_n))') \wedge t_1^* \wedge \cdots \wedge t_n^*$;

- $(t[l_1 \Rightarrow e_1, \cdots, l_n \Rightarrow e_n])^* \equiv t^* \wedge \alpha_1^* \wedge \cdots \wedge \alpha_n^*$, where for every e_i, either
  - e_i is a term and α_i^* is $e_i^* \wedge l_i(t', e_i')$; or
  - e_i is a collection $\{t_1^i, \cdots, t_{n_i}^i\}$ of terms, and α_i^* is $(t_1^i)^* \wedge l_i(t', (t_1^i)') \wedge \cdots \wedge (t_{n_i}^i)^* \wedge l_i(t', (t_{n_i}^i)')$;
- $(p(t_1, \cdots, t_n))^* \equiv t_1^* \wedge \cdots \wedge t_n^* \wedge p(t_1', \cdots, t_n')$.

It can be proved by induction that, given any atomic formula α, $\mathbf{M} \models \alpha[s]$ iff $\mathbf{M}^* \models \alpha^*[s]$ for any semantic structure $\mathbf{M}$ and any variable assignment s. QED.
Example 2: The atomic formula

determiner: the[num ⇒{singular, plural}, def ⇒definit]

can be transformed into

determiner(the) ∧
object(singular) ∧ num(the, singular) ∧
object(plural) ∧ num(the, plural) ∧
object(definite) ∧ def(the, definite)

Recall that a term of the form *object:* t can be abbreviated as t.

4 Specification and Computation of Complex Objects

We are interested in a logic programming approach to complex objects. That is, complex objects can be specified by facts and rules in a logic, and complex object reasoning or computation can be carried out as logic inferencing. Given a language of objects, a clausal subset of formulas is defined as follows:

- A literal is either an atomic formula or the negation of an atomic formula.
- A clause is a disjunction of literals

  $$L_1 \vee L_2 \vee \cdots \vee L_n$$

  in which all variables are implicitly universally quantified at the outmost level.
- A definite clause is a clause containing exactly one positive literal, i.e., $A \vee \neg B_1 \vee \cdots \vee \neg B_m$ $(m \geq 0)$, which is also written as

  $$A :- B_1, \cdots, B_m.$$

  where A, $B_1, \cdots, B_m$ are all atomic formulas.
- A negative clause is a clause containing no positive literals, i.e., $\neg B_1 \vee \cdots \vee \neg B_m$, which is also written as

  $$:- B_1, \cdots, B_m.$$

  A negative clause is also called a query or a goal.

In the definition of languages of objects, we have assumed, for simplicity, that each language has a countable partially ordered set of type symbols with a greatest element *object*. The only assumption we actually need is that each language has the greatest type *object*, which is a supertype of any other type. And we would like to have the user be able to specify the ordering among other type symbols. Subtype declarations are therefore introduced:

- A subtype declaration is of the form

  $$\iota_1 \leq \iota_2.$$

  where ι_1 and ι_2 are type symbols.

A program is a finite set of subtype declarations and definite clauses.
Example 3: The following program defines objects of type *noun_phrase*.

```
name: john.
name: bob.

determiner: the[num ⇒{singular, plural},
                def ⇒definite].
determiner: a[num ⇒singular, def ⇒indef].
determiner: all[num ⇒plural, def ⇒indef].

noun: student[num ⇒singular].
noun: students[num ⇒plural].

proper_np: X[pers ⇒3, num ⇒singular,
             def ⇒definite] :-
        name: X.
common_np: np(Det, Noun)[pers ⇒3,
             num ⇒N, def ⇒D] :-
        determiner: Det[num ⇒N, def ⇒D],
        noun: Noun[num ⇒N].

proper_np ≤ noun_phrase.
common_np ≤ noun_phrase.
```

And we may pose a query like

```
:- noun_phrase: X[num ⇒plural].
```

and obtain two answers for X: np(the, students) and np(all, students).

We have shown in the previous section that every atomic formula in a language of objects can be transformed into an equivalent conjunction of first-order atomic formulas. Thus every clause of the form

$$L_1 \vee \cdots \vee L_n$$

can be transformed into a disjunction of conjunctions (or the negation of conjunctions) of first-order atomic formulas

$$S_1(A_1^1 \wedge \cdots \wedge A_{k_1}^1) \vee \cdots \vee S_n(A_1^n \wedge \cdots \wedge A_{k_n}^n)$$

where S_i is either $\neg$ if L_i is a negative literal or empty if L_i is a positive literal. The resulting clause is called a gener-

alized clause. If the original clause is a definite clause, the resulting clause is also called a generalized definite clause. Obviously, a generalized (definite) clause can be further transformed into a finite number of first-order (definite) clauses.

Every subtype declaration of the form

$$\iota_1 \leq \iota_2.$$

can be transformed into a first-order clause of the form

$$\iota_2(X) \text{ :- } \iota_1(X).$$

Given a program of objects consisting of a finite set of subtype declarations and definite clauses, we can obtain a generalized logic program composed of a finite set of first-order clauses and generalized definite clauses. And this generalized logic program can be further transformed into a first-order logic program consisting of only first-order definite clauses.

There is still one technicality in transforming a logic program of objects into an equivalent first-order logic program because of the type hierarchy. We assume that there is a type *object* which is a supertype of any other type. If we translate this assumption into first-order axioms, we could possibly get an infinite number of first-order clauses because of the possibly infinite number of type symbols. Fortunately every logic program of objects contains only a finite number of type symbols. And we need to incorporate type axioms for only those type symbols occurring in the program. That is, for every type symbol ι occurring in the object program, we add a first-order clause

$$object(X) \text{ :- } \iota(X).$$

to the corresponding generalized logic program (and hence also to the final first-order logic program).

For instance, the above program can be transformed into the following generalized logic program. Recall that if the type of a term is *object*, the type can be omitted.

```
object(X) :- name(X).
object(X) :- determiner(X).
object(X) :- noun(X).
object(X) :- proper_np(X).
object(X) :- common_np(X).
object(X) :- noun_phrase(X).

name(john).
name(bob).

determiner(the),
        object(singular), num(the, singular),
        object(plural), num(the, plural),
        object(definite), def(the, definite).
determiner(a), object(singular), num(a, singular),
                object(indef), def(a, indef).
determiner(all), object(plural), num(all, plural),
                object(indef), def(all, indef).

noun(student), object(singular),
                num(student, singular).
noun(students), object(plural),
                num(students, plural).

proper_np(X), object(3), pers(X, 3),
        object(singular), num(X, singular),
        object(definite), def(X, definite) :-
        name(X).
common_np(np(Det, Noun)),
            object(Det), object(Noun),
            object(3), pers(np(Det, Noun), 3),
            object(N), num(np(Det, Noun), N),
            object(D), def(np(Det, Noun), D)
  :-
    determiner(Det), object(N), num(Det, N),
    object(D), def(Det, D),
    noun(Noun), object(N), num(Noun, N).

noun_phrase(X) :- proper_np(X).
noun_phrase(X) :- common_np(X).
```

which can be easily transformed into first-order logic programs by splitting each generalized definite clause into a finite number of first-order definite clauses. And the query would be

```
:- noun_phrase(X), object(plural),
                num(X, plural).
```

Notice that multiple occurrences of the same variable in the head are independent. For example, the rule for proper_np can be split into several clauses like

```
proper_np(X) :- name(X).
pers(X, 3) :- name(X).
num(X, singular) :- name(X).
...
```

in which every occurrence of X is universally quantified with respect to each clause.

There are two important implications of this transformation into first-order definite clauses. First, model-theoretic results in deductive databases and logic programming can be readily applied. (Negation can also be added although we do not include it in this paper.) Second, known query evaluation techniques, including both bottom-up and top-down methods, can be used for computation of complex objects. Because of the structure of complex objects, each rule of complex object specification naturally corresponds to a generalized or multi-head first-order clause. Therefore, in bottom-up computation, each successful evaluation of the body may produce multiple results.

However, it is also obvious from the above example that the resulting first-order logic program may have certain redundancies, especially in typing predicates. Most of these redundancies can be eliminated by static program analysis. We consider two cases for generalized logic programs:

1. If both $\iota_1(X)$ and $\iota_2(X)$ appear in the head or the

body of a generalized definite clause, and $\iota_1 \leq \iota_2$, then $\iota_2(X)$ can be deleted;

2. If $\iota_1(X)$ appears in the head and $\iota_2(X)$ in the body of the same generalized definite clause, and $\iota_2 \leq \iota_1$, then $\iota_1(X)$ in the head can be deleted.

After these two cases are eliminated, the definition for *common_np* would become

```
common_np(np(Det, Noun)), object(3),
                  pers(np(Det, Noun), 3),
                  num(np(Det, Noun), N),
                  def(np(Det, Noun), D) :-
        determiner(Det), object(N),
        num(Det, N), object(D), def(Det, D),
        noun(Noun), num(Noun, N).
```

There are other redundancies whose detection requires a little bit more complicated program analysis. For example, there are many redundant clauses for *object* which can be eliminated. The type *object* is special in the sense that it is a supertype of every other type. It can be used to inquire about the existence of an object in the database. If we consider the corresponding first-order logic program for a program of objects, then the meaning of *object* is actually the set of all ground terms in the success set. So *object* is essentially the active domain which includes every individual object in the database.

An interesting alternative is to consider a direct implementation of complex object reasoning without translating complex object specification into first-order logic programs. This is based on the following observation. The syntax of complex objects allows the user to cluster component objects together according to specific applications. And it is very likely that the user would also pose queries that way. Reasoning directly over complex objects may allow the system to take advantage of such clustering information provided by the user. This seems to be the case, especially when most labels are functional or single-valued. For example, given a program

```
path: p1[src ⇒a, dest ⇒b].
path: p2[src ⇒c, dest ⇒d].
```

in which all labels are functional, and a query

```
:- path: X[src ⇒S, dest ⇒D].
```

we can evaluate the query by unifying it with each fact and all the two sets of answers will be obtained for X, S, and D. Notice that if the above program is first transformed into the corresponding first-order logic program

```
object(X) :- path(X).
path(p1). object(a). src(p1, a).
          object(b). dest(p1, b).
path(p2). object(c). src(p2, c).
          object(d). dest(p2, d).
```

and the query would be

```
:- path(X), object(S), src(X, S),
        object(D), dest(X, D).
```

whose direct evaluation using SLD resolution directly would be very inefficient.

For multi-valued labels, query evaluation would be more complex. Suppose that we have two facts:

```
path: p[src ⇒a, dest ⇒b].
path: p[src ⇒c, dest ⇒d].
```

and a query

```
:- path: p[src ⇒a, dest ⇒d].
```

The query should succeed according to our semantics. However, naive evaluation using unification will fail. The problem is that we need to solve part of the query at one time, take the residual and then proceed. In this example, we first try to solve label src using the first fact, and get the residual

```
:- path: p[dest ⇒d].
```

which can then be solved using the second fact. (Recall that labels of a term are independent, while arguments of a predicate are associated together.)

For extensional databases, we may merge all information about an object together. For the above example, we may have a single fact:

```
path: p[src ⇒{a, c}, dest ⇒{b, d}].
```

and the query can be solved by checking certain ordering over complex object descriptions [6]. However, in intensional databases, it is conceivable that there may be several rules, each of which deals with partial information about the same object. And we cannot simply merge these rules together.

Another aspect which may be more easily handled by direct reasoning over complex objects is types. Using order-sorted resolution may be more efficient in dealing with inheritance hierarchies [4,11].

5 Multi-Valued Labels and Sets

Set manipulation seems to be accepted as an important aspect of any system for complex objects. It is, however, not yet clear how sets should be supported in a simple logic framework. Our C-logic of complex objects is first-order and thus does not have set values. On the other hand, multi-valued labels do allow the user to use the concept of sets (in useful but restricted ways) to model complex objects. For example, we might have a fact

```
person: john[children ⇒{bob, bill, joe}].
```

and a query

```
:- person: john[children ⇒{X, Y}].
```

According to our semantics, labels are essentially binary predicates. Both the fact and the query can be transformed into the following:

person: john[children ⇒bob,
children ⇒bill, children ⇒joe].

:- person: john[children ⇒X, children ⇒Y].

And both X and Y can be bound to each of *bob, bill, joe* to make the query succeed.

Rather than thinking of a label as a binary predicate, the user may consider a label *intuitively* as a set function. That is, a label maps each object identity to a (possibly empty or infinite) set of object identities. So ⇒ can be understood as either "containing as a subset" if a collection of terms is followed, or "containing an element" if a single term is followed. For instance, the fact in the above example can be viewed as saying that *children* maps *john* to a set of objects which contains {bob, bill, joe} as a subset. And the original query can be considered as asking for a set {X, Y} which is contained in the set of *children* corresponding to *john*. Furthermore, by passing *john* around, the set associated with *john* by *children* (in the intuitive sense) can be indirectly accessed through object *john*. If complex objects are defined by rules, then definitions in separate rules support set union. And unification supports certain aspects of set intersection. Pragmatically, these uses may be able to satisfy most of what a user wants from set manipulation. The only aspect that is lacking in our logic is the ability to return a set value and to check whether two sets are equal (i.e. set unification).

The idea of using multi-fields for set manipulation was originally proposed by Maier in [22]. But the formal semantics of multi-fields was not specified. Set union through separate rules appeared in CCO [6] by Bancilhon and Khoshafian, although only finite sets can be constructed. This idea was further extended by Kifer and Wu in [18], where each set is associated with an object identity. We have given a formal first-order semantics for set manipulation through multi-valued labels. This shows that without using higher-order logic, we can still support many useful aspects of set manipulation.

6 Conclusion and Future Work

We have presented a simple and flexible framework for reasoning about complex objects. It supports fundamental features of complex objects such as object identity, multi-valued labels and a dynamic notion of types. Although the semantics is first-order, the logic itself encourages structured description of complex obejcts and provides many useful aspects of set manipulation through multi-valued labels. Static type constraints and single-valued labels are not built into the logic but can be added on top of it if needed. This makes the logic much simpler, more flexible and easier to implement.

There are several issues worthy of further exploration. First, because of the heterogeneous structure of complex objects, there are problems of how to store complex objects, how to cluster components of a complex object together, and how to efficiently reason about complex objects. Work on some of these issues has been reported [16,19]. The simplicity and flexibility of our scheme makes it a suitable paradigm for pursuing these problems. Second, in addition to object data, there is also meta-data or schema information in databases. This meta-data usually includes constraints over database states, such as domain constraints and functionality constraints. How to extend C-logic to incorporate these constraints and to reason about them is another interesting problem to be further studied.

Acknowledgements

Our work obviously depends crucially on Maier's O-logic. We have benefitted greatly from discussions in a series of joint meetings with Michael Kifer, James Wu, Chyouhwa Chen, T. Krishnaprasad, Esther Shilcrat and Jiyang Xu. And we thank David Maier and his group for many helpful comments on a draft of this paper. This work is supported in part by the National Science Foundation under grant number DCR-8319966.

References

[1] Abiteboul S. and Beeri C., On the Power of Languages for the Manipulation of Complex Objects, *Draft,* April 1987.

[2] Abiteboul S. and Grumbach S., COL: A Logic-Based Language for Complex Objects, in: *Proc. Workshop on Database Programming Languages,* Roscoff, France, September 1987, pp. 253-276.

[3] Ait-Kaci H. and Nasr R., Logic and Inheritance, in: *13th ACM Symp. on Principles of Programming Languages,* Florida, January 1986, pp. 219-228.

[4] Ait-Kaci H. and Nasr R., LOGIN: A Logic Programming Language with Built-in Inheritance, em Journal of Logic Programming 3(1986), pp. 185-215.

[5] Buneman P. and Atkinson M., Inheritance and Persistence in Database Programming Languages, in: *Proc. of ACM SIGMOD'86,* Washington, D.C., May 1986, pp. 4-15.

[6] Bancilhon, F. and Khoshafian S., A Calculus for Complex Objects, in: *Proc. of 5th ACM Symp. on Principles of Database Systems,* Cambridge, Massachusetts, March 1986, pp. 53-59.

[7] Beeri C., Naqvi S., Shmueli O. and Tsur S., Sets and Negations in a Logic Database Language

(LDL), in: *Proc. ACM Conf. PODS*, San Diego, March 1987, pp. 21-37.

[8] Beeri C., Nasr R. and Tsur S., Embedding ψ-terms in a Horn-Clause Logic Language, MCC Technial Report, February 1988.

[9] Cardelli L., A Semantics of Multiple Inheritance, in: *Semantics of Data Types, LNCS 173*, 1984, pp. 51-67.

[10] Debray S.K. and Warren D.S., Detection and Optimization of Functional Computations in Prolog, in: *Third International Conference on Logic Programming*, ed. Ehud Shapiro, London, July 1986, pp. 490-504.

[11] Huber M. and Varsek I., Extended Prolog for Order-Sorted Resolution, in: *Proc. 1987 IEEE Symp. on Logic Programming* 1987, pp. 34-43.

[12] Hull R., A Survey of Theoretical Research on Typed Complex Database Objects, in: *Databases*, ed. J. Paredaens, Academic Press (London), 1987, pp. 193-256.

[13] Hull R., Four Views of Complex Objects: A Sophisticate's Introduction, *Draft*, May 1988.

[14] Kuper G., Logic Programming with Sets, in: *Proc. 6th ACM Conf. on PODS*, San Diego, 1987, pp. 11-20.

[15] Kuper G., An Extension of LPS to Arbitrary Sets, in: *IBM Research Report*, 1987.

[16] Ketabchi M.A. and Berzins V., Mathematical Model of Composite Objects and Its Application for Organizing Engineering Databases, *IEEE Trans. Software Engineering*, vol. 14, no. 1, January 1988, pp. 71-84.

[17] Khoshafian S.N. and Copeland G.P., Object Identity, in: *OOPSLA'86*, 1986, pp. 406-416.

[18] Kifer M. and Wu J., A Logic for Object-Oriented Logic Programming (Maier's O-logic Revisited), in: *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1989*, to appear.

[19] Kim W., Chou H.-T. and Banerjee J., Operations and Implementation of Complex Objects, *IEEE Trans. Software Engineering*, vol. 14, no. 7, July 1988, pp. 985-995.

[20] Krishnamurthy R. and Naqvi S., Towards a Real Horn Clause Language, *MCC Technical Report No. ACA-ST-077-88*, March 1988.

[21] Kuper G.M. and Vardi M.Y., A New Approach to Database Logic, in: *Proc. ACM SIGACT-SIGMOD Symp. on PODS*, 1984, pp. 86-96.

[22] Maier D., A Logic for Objects, in: *Preprints of Workshop on Foundations of Deductive Database and Logic Programming*, ed. Jack Minker, Washington DC, August 1986.

[23] Roth M.A., Korth H.F. and Silberschatz A., Extended Algebra and Calculus for $\neg$1NF Relational Databases, Technical Report TR-84-36, University of Texas at Austin, 1984.

[24] Xu J. and Warren D.S., A Type Inference System for Prolog, in: *Proc. 5th Internat. Conf and Symp. on Logic Programming* 1988, pp. 604-619.