ILOG: Declarative Creation and Manipulation of Object Identifiers*

(Extended Abstract)

Richard Hull[†]

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782
USA
hull@cse.usc.edu

Masatoshi Yoshikawa[‡]

Department of Information and
Communication Sciences
Kyoto Sangyo University
Kamigamo, Kita-ku, Kyoto 603
Japan
yosikawa@kyoto-su.ac.jp

Abstract: This paper introduces ILOG$^{(\neg)}$, a declarative language in the style of (stratified) datalog$^{(\neg)}$, which can be used for querying, schema translation, and schema augmentation in the context of object-based data models. The semantics of ILOG$^{(\neg)}$ is based on the use of Skolem functors, and is closely related to semantics for object-based data manipulation languages which provide mechanisms for explicit creation of object identifiers (OIDs). A normal form is presented for ILOG$^{\neg}$ programs not involving recursion through OID creation, which identifies a precise correspondence between OIDs created in the target, and values and OIDs in the source. The expressive power of various sublanguages of ILOG$^{\neg}$ is shown to range from a natural generalization of the conjunctive queries to the object-based context, to a language which can specify all computable database translations (up to duplicate copies). The issue of testing *validity* of ILOG$^{(\neg)}$ programs translating one semantic schema to another is studied: cases are presented for which several validity issues (e.g., functional and/or subset relationships in the target schema) are decidable; and other cases are presented for which they are undecidable. Non-recursive ILOG is rich enough to simulate some schema translation languages based on local structural manipulation, and non-recursive ILOG$^{\neg}$ can simulate the core of the OOAlgebra of [Day89], and of several other translation languages of the systems literature.

*This research supported in part by NSF grant IRI-8719875.

[†]Research by this author also supported in part by NSF grant INT-8817874 and a grant from AT&T

[‡]Research by this author was performed while visiting the Computer Science Department of the University of Southern California. This author was supported in part by Yazaki Memorial Foundation for Science & Technology.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1 Introduction

Object-based data models (both semantic and object-oriented) have received a great deal of attention over the past few years. A fundamental problem in this area is the development of data manipulation languages for object-based models which have a mathematical foundation as rigorous as the relational calculus and algebra. A key difference between the well-understood relational context and the object-based context is the presence, in the latter case, of *object identifiers* (*OIDs*) (or *surrogates*), which correspond to real and conceptual objects "in the world". This paper introduces the family of ILOG$^{(\neg)}$ languages; these are declarative languages for querying, schema translation, and schema augmentation which generalize datalog$^{(\neg)}$ and which support explicit OID creation and manipulation. The paper also presents a number of results analyzing these languages, and comparing them with previously published data translation languages.

An important impetus for the development of ILOG$^{(\neg)}$ was the introduction, in the papers on LDM [KV84] and IQL [AK89], of formal data models and query languages which incorporate object identity in an explicit fashion. The IQL paper in particular demonstrated, among other things, how the presence of OIDs and recursion can lead to query languages with extremely rich restructuring capabilities, and expressive power close to that of Turing machines. The present investigation continues with this focus on the impact of explicit OIDs on fundamental database issues. A key difference, however, is the semantics used for OID creation: IQL uses a variation of the invention rules of detDL [AV88a, AV88b], while ILOG$^{(\neg)}$ uses the observation of Maier [Mai86], refined in [KW89, CW89], that OID creation can be simulated by associating Skolem functors with datalog rules. (A specific theoretical difference between languages resulting from the two approaches to OID creation is exhibited in Example 7.6.) The use of Skolem functors makes it natural in ILOG$^{(\neg)}$ to use the conventional (stratified) least fixpoint semantics of logic programming. This permits a transfer of techniques from logic programming and datalog to the study of ILOG$^{(\neg)}$. Furthermore, this paper establishes a normal form (Proposition 7.1) for ILOG$^{\neg}$ programs not involving recursion through OID creation. This provides a formal mechanism for representing the correspondence between OIDs created in a target database and the OIDs and values of the source database. The normal form is a natural and useful tool for the development of many of the results of this paper, and also appears useful in the areas of update propagation and query optimization.

The ILOG$^{(\neg)}$ framework is designed to permit a focused study on the impact of explicit OIDs on database issues, and assumes a relatively simple context. For example, unlike the approaches of both [AK89] and [Mai86, KW89, CW89], the framework here is based on a simple semantic database model (a subset of IFO [AH87] called IFO^{-}) extended to incorporate object identity. In its current form, this model does not permit sets as[1] "first-class citizens", and has a natural simulation in the relational model. Additionally, as with logic programming and datalog, ILOG$^{(\neg)}$ uses (in the terminology of, e.g., [Jac89]) *descriptive* typing: the typing is not implicit to the language, but a type-checking inference mechanism based on the source and target schemas can be established. This contrasts with IQL, OODAPLEX [Day89], and OOAlgebra [Day89], which use *prescriptive* typing: these languages support (and require) the explicit declaration of types for variables and program structures. Finally, the ILOG$^{(\neg)}$ languages focus on data manipulation, and do not include mechanisms for specifying the target schema. It is assumed that a schema definition language is used for that purpose.

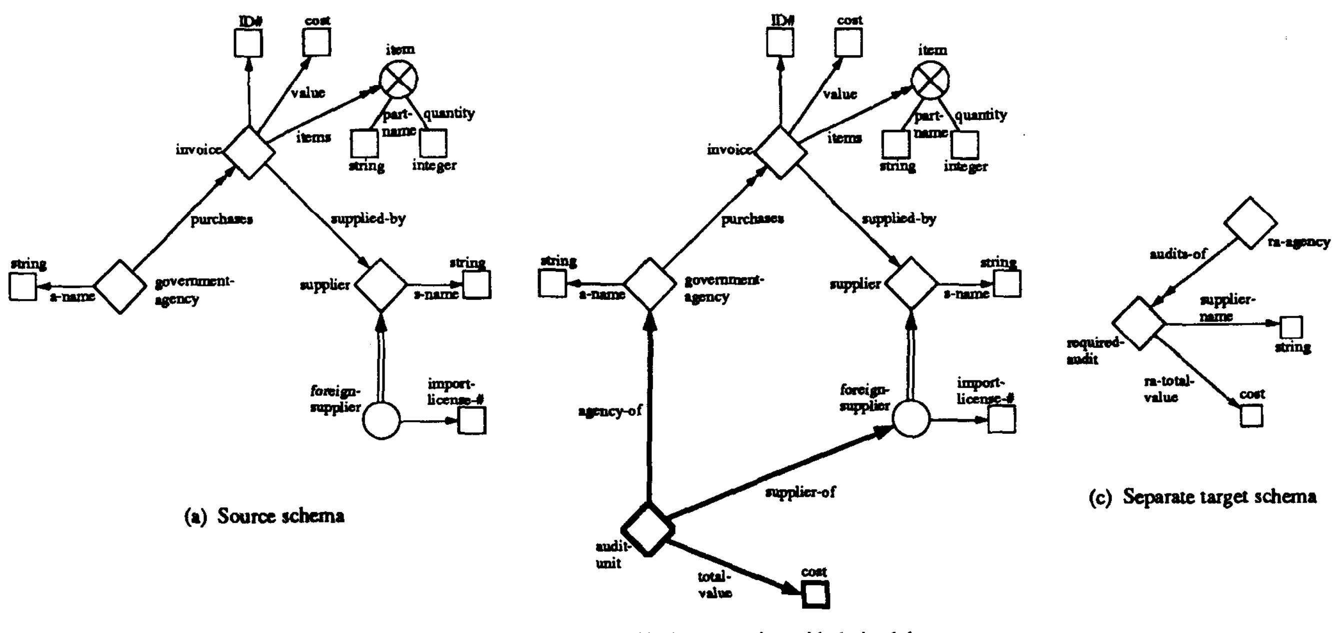
This paper presents theoretical results for ILOG$^{(\neg)}$ in two directions: (a) expressive power and complexity, and (b) testing the "validity" of schema translations expressed in ILOG$^{(\neg)}$. The first topic considers seven syntactically defined ILOG languages. Six of these stem from two dimensions: (i) the presence or absence of stratified negation, and (ii) permitting full recursion, permitting "weak" recursion (i.e., not through OID creation), and prohibiting recursion. It is shown that all of these languages have distinct expressive powers, with nonrecursive ILOG without negation (nrecILOG) having power closely related to the conjunctive queries, and fully recursive ILOG with negation (ILOG$^{\neg}$) "almost" capable of specifying all computable database translations. (The proviso here stems from the inability of ILOG$^{\neg}$ to "remove copies", a difficulty initially observed in connection with IQL.) The weakest language, nrecILOG, can simulate the schema translation languages of the schema integration methodology presented in [BM81, Mot87], and nonrecursive ILOG with stratified negation (nrecILOG$^{\neg}$) subsumes the cores of the OOAlgebra and the translation languages of the Federated architecture [HM85] and the integration methodology of [DH84]. It is also shown that weakly recursive ILOG$^{(\neg)}$ (wrecILOG$^{(\neg)}$) has the same data complexity as datalog$^{(\neg)}$ (and thus lies between NLOGSPACE and PTIME). The seventh ILOG variant is ILOG$^{\pm}$, a practically motivated language for which a prototype has been implemented [HWW90]. In formal terms, the core of ILOG$^{\pm}$ is a generalization of nrecILOG which permits the use of the full relational calculus on source relations. It is shown here that the core of ILOG$^{\pm}$ is equivalent to nrecILOG$^{\neg}$.

An ILOG$^{(\neg)}$ program P mapping from schema S to schema T is *valid* if for each instance I of S, the result of applying P (if the computation terminates) is an instance of T. (The result might fail to be an instance by violating functional, subset, or disjointness relationships required by T.) It is shown here that validity is decidable in EXPSPACE for IFO^{-} schemas and nrecILOG. In particular, this result holds if (a) $\neq$ is permitted in rule bodies, or (b) some attributes in the source and/or target schemas are required to be total. As a result, validity is decidable for the language of [BM81, Mot87]. (Decidability of validity remains open if both (a) and (b) are present.) Validity for the richer variants of ILOG$^{(\neg)}$ (and thus, OOAlgebra, OODAPLEX and the languages of [DH84, HM85]) is shown to be undecidable.

In the context of schema translation, the formal semantics of ILOG$^{(\neg)}$ programs is defined by (a) Skolemizing the rules of the program, (b) taking the minimal model of the program as in logic programming (stratified model if negation is present), and finally, (c) (non-deterministically) replacing the non-atomic terms in the result by (distinct, essentially new) OIDs. A semantics for ILOG$^{(\neg)}$ is also presented for the case of schema augmentation, i.e., defining derived data. The ILOG$^{(\neg)}$ syntax "hides" the Skolem functors from the user, thus bringing ILOG$^{(\neg)}$ closer to some other, existing mechanisms for OID manipulation in the literature (e.g., Smalltalk-80 [GR83], OODAPLEX, OOAlgebra). Although not explored in this paper, it appears that the Skolem-functor based semantics of ILOG$^{(\neg)}$ can be used to provide a natural and rigorous basis for update propagation and query optimization. In particular, the normal form result of Proposition 7.1 for wrecILOG$^{\neg}$ programs provides a mechanism whereby the created OIDs can be viewed as Skolem terms, and thus carry within them information about how/why they were created. This approach can be extended to many practical schema translation languages and OOAlgebra, because they are subsumed by wrecILOG$^{\neg}$.

As noted above, ILOG$^{(\neg)}$ in its current form can be used in connection with any object-based model for which there is a natural simulation by the relational model. A natural extension of ILOG$^{(\neg)}$ could be made to provide mechanisms for creating and manipulating sets, as in COL [AG88],

[1]Speaking informally, a model incorporates sets as "first-class citizens" if sets can be formed and manipulated independent of the use of tuples of pre-existing atomic values or objects which refer to them. For example, nested relations in PNF [RKS88, AB86] do not permit sets as first-class citizens, because each set occurring in a PNF nested relation can be uniquely identified by a tuple of atomic values. The analogous statement holds for the FDM although it supports multi-valued attributes. Examples of models with sets as first-class citizens include nested relations, complex objects, SDM, LDM, IFO, and the models underlying LDL, COL and IQL (see [Hul89] for further discussion).



(a) Source schema

(b) Augmentation with derived data

(c) Separate target schema

Figure 1: Example schemas

LDL [NT89], IQL, or the langauge of [Abi89]. As shown in Exapmle 7.5, the inclusion of a "grouping" construct (which is present in IQL) into $\mathrm{ILOG}^{\neg}$ would yield a strictly more powerful language, even if the source and target schemas do not include the set construct.

Section 2 presents brief examples of nrecILOG, and the intuition behind the formal semantics of $\mathrm{ILOG}^{(\neg)}$ based on Skolem functors. Section 3 reviews concepts and notation needed for the formal development. Section 4 articulates some key differences between semantic data models and models in which object identity is an explicit construct. In particular, the section introduces a new definition of *instance* for semantic schemas, which formally captures the intuition that OIDs, in and of themselves, carry no information. Section 5 introduces the formal syntax and semantics of $\mathrm{ILOG}^{(\neg)}$, and Section 6 presents the equivalence of this semantics to one based on "explicit OID creation", a variation of the semantics used for IQL and most practical data manipulation languages for object-based models. Section 7 presents normal forms for nrecILOG and $\mathrm{wrecILOG}^{\neg}$; considers the expressive power of ILOG languages; and provides comparisons with the semantics and power of both IQL and practical schema translation languages. Section 8 considers validity testing. Due to space limitations, much of the discussion is terse, and proofs are omitted or sketched. Details may be found in the full paper [HY91].

2 Example and motivation

This section presents an extended example which illustrates how ILOG can be used to create and manipulate OIDs, and gives the intuition behind the formal semantics of ILOG. The example uses a simple subset of ILOG in which recursion and negation are not permitted. (The example also illustrates a form of aggregate operator, although aggregates are not considered in the theoretical treatment of ILOG which follows.) The presentation in this section is largely informal; precise definitions are given in Sections 3, 5 and 6 below.

ILOG can be used for (a) defining derived data, (b) specifying a translation from one schema to another, and (c) expressing queries. In the first case, the original schema is *augmented*, and the new data may refer to the input data. In the second case the new data is presumed to be independent of the source data. Interpreted in a broad sense, queries can involve both the definition of derived data and/or the translation of (selected portions of) the source to a new schema. In all cases, there are two aspects to the specification: describing the new schema (or new schema components), and describing how the new schema (components) are to be populated. Several languages for specifying schema definition (augmentation) have been developed; we do not consider that further here. The ILOG languages focus on the second aspect.

Consider the semantic data model schema shown in Figure 1(a). (This is an IFO^- schema as defined in Section 3; the diagramatic conventions are essentially those of [AH87, HK87], and similar to those of [Shi81].) This schema models hypothetical data concerning purchases made by governmental agencies. The schema includes *abstract types* for `government-agency`, `invoice`, and `supplier`, and a *subtype* of `supplier` called `foreign-supplier`. *Single-valued attributes* include `a-name` mapping each government agency to a string, and `value` mapping each invoice to a cost (viewed here as an atomic, printable type). The *multi-valued attribute* `purchases` maps each government agency to a set of invoices, which represent purchases made by that agency. The *aggregation* (or *tuple*) *construct* indicates that each `item` is an ordered pair, consisting of a `part-name` and a `quantity`.

ILOG exploits the natural correspondence between semantic schemas (in which sets do not occur as "first-class citizens") and relational schemas. Under this correspondence, abstract types and subtypes map to unary relations, and (single- and multi-valued) attributes and aggregations map to relations of the appropriate arity. (For example, the a-name attribute on government-agency maps to a binary relation *a-name* satisfying the functional dependency $1 \to 2$; and the items attribute on invoice maps to a ternary relation *items*, with no associated functional dependencies.) The specific naming convention used is left to the system designer; we use *ad hoc* naming here.

Example 2.1: We first augment the schema of Figure 1(a) to form the schema of Figure 1(b) (the added components are highlighted), and use ILOG to populate the augmentation. In particular, we create a new entity set called audit-unit – each audit-unit is a new conceptual object corresponding to an agency-supplier pair (a, s) where a has at least one invoice with supplier s, and s is a foreign supplier. Intuitively, each audit-unit can serve as a locus for data concerning audits of such agency-supplier pairs. (In this example, only one audit-unit OID is created for a pair (a, s), regardless of how many invoices relate a with s; different rules could be used to obtain other policies for OID creation.) We also define two single-valued attributes which associate these newly created objects to agencies and suppliers, and a single-valued attribute which gives the sum of values of all invoices associated with the agency-supplier pair associated with the audit-unit. The following ILOG program accomplishes this, with output relations *audit-unit*, *agency-of*, *supplier-of*, and *total-of*:

$$\begin{array}{lll}
\textit{int-aud-un}(*, a, s) & \leftarrow & \textit{purchases}(a, i), \\
\multicolumn{3}{r}{\textit{supplied-by}(i, s), \textit{foreign-supplier}(s)} \\
\textit{audit-unit}(u) & \leftarrow & \textit{int-aud-un}(u, a, s) \\
\textit{agency-of}(u, a) & \leftarrow & \textit{int-aud-un}(u, a, s) \\
\textit{supplier-of}(u, s) & \leftarrow & \textit{int-aud-un}(u, a, s) \\
\textit{total-value}(u, \mathrm{sum}\langle v\rangle) & \leftarrow & \textit{int-aud-un}(u, a, s), \\
\multicolumn{3}{r}{\textit{purchases}(a, i), \textit{supplied-by}(i, s), \textit{value}(i, v)}
\end{array}$$

Intuitively, execution of this program results in the creation of (new) OIDs for each (a, s) pair satisfying the conditions of the body of the first rule given above. The *intermediate* relation *int-aud-un* is used to "create" each such OID, and to "hold" its correspondence to the *witness* (i.e., tuple of values and OIDs which lead to its creation.) As with conventional datalog, variables in the rule body not occurring in the head (e.g., the variable i ranging over invoices) are viewed as existentially quantified within the body.

The remaining rules are used to describe how the four components added to the schema are to be populated. Although we do not include a formal study of aggregate operations in this paper, we have included the last rule to illustrate how certain aggregate functions can be naturally incorporated into the ILOG (and datalog) framework (alternative approaches are considered in [NT89, She90]). Speaking informally, the semantics of this rule is as follows: For each u, the set $S(u)$ of tuples (a, s, i, v) satisfying the rule body is obtained. A projection of $S(u)$ onto the v coordinate is performed to obtain a *multiset* of v-values. The sum operator is applied to this bag to obtain the value associated with u by *total-value*. □

As a second example, we describe an ILOG program that can be used to translate (some of the) data from the original schema to a completely separate schema, shown in Figure 1(c). Importantly, no OIDs used in the source schema will be permitted in the instances computed for the target schema.

Example 2.2: In this translation we build up a set of audit-units for which the total value is at least one million dollars; we assume that an audit is required for each such audit-unit. We include an abstract type for the agencies having required audits, the attribute audits-of mapping each such agency to the set of related audit-units, an attribute giving the name of the associated supplier, and again give the total-value of the audit-unit. (We use a different attribute name, because the attribute used here is a restriction of the attribute used before.) The ILOG program to accomplish this translation includes the first and last rules from the program given above, and also the six rules given now. The output of this program is given by the relations *required-audit*, *ra-agency*, *audits-of*, *supplier-name*, and *ra-total-value*.

$$\begin{array}{lll}
\textit{required-audit}(u) & \leftarrow & \textit{int-aud-un}(u, a, s), \\
 & & \quad \textit{total-value}(u, v), v \geq 10^6 \\
\textit{int-ra-agency}(*, a) & \leftarrow & \textit{int-aud-un}(u, a, s), \\
 & & \quad \textit{required-audit}(u) \\
\textit{ra-agency}(a') & \leftarrow & \textit{int-ra-agency}(a', a) \\
\textit{audits-of}(a', u) & \leftarrow & \textit{int-ra-agency}(a', a), \\
\multicolumn{3}{r}{\textit{int-aud-un}(u, a, s), \textit{required-audit}(u)} \\
\textit{supplier-name}(u, n) & \leftarrow & \textit{int-aud-un}(u, a, s), \\
\multicolumn{3}{r}{\textit{required-audit}(u), \textit{a-name}(s, n)} \\
\textit{ra-total-value}(u, v) & \leftarrow & \textit{required-audit}(u), \\
 & & \quad \textit{total-value}(u, v)
\end{array}$$

(In this example we created new OIDs for *ra-agency*. As detailed in Remark 5.3 below, this program can be made more succinct by permitting the explicit use of OIDs from the source in the target. However, since this program is considered as a translation specification, the "association" of these OIDs to the source instance will be lost.) □

As detailed in Section 5, ILOG$^{\neg}$ supports full recursion and stratified negation. Also, although not done in this paper, the ILOG languages could be generalized to permit the direct use of single- and multi-valued attributes in the style of [AH88, AG88] (e.g., permitting the use of atoms '$\textit{supplier-name}(s) = n$' and '$u \in \textit{audits-of}(a)$').

We conclude this section with intuitive remarks concerning the formal semantics of ILOG. A fundamental influence on the development of ILOG is found in the so-called "alphabet logics": O-logic [Mai86], 'O-logic-revisited' [KW89], and C-logic [CW89]. In particular, the premise that OIDs are essentially terms built using Skolem functors appears to have initially surfaced in these papers. (The more basic intuition that objects are naturally created from aggregations (tuples) of values and objects may be found in semantic model schema translation languages described in, e.g., [BM81, HM85, Day89].) To illustrate this point, recall the rule of Example 2.1 for defining *int-aud-un*. Following [KW89, CW89], we note that in first-order logic, this rule could be rewritten as

$$\forall a\ \forall s\ \exists u\ \forall i\ [\mathit{int\text{-}aud\text{-}un}(u,a,s) \leftarrow \mathit{purchases}(a,i), \mathit{supplied\text{-}by}(i,s), \mathit{foreign\text{-}supplier}(s)\ ]$$

Following [KW89, CW89], a Skolem functor[2] is now introduced:

$$\forall a\ \forall s\ \forall i\ [\mathit{int\text{-}aud\text{-}un}(f_{a\text{-}u}(a,s),a,s) \leftarrow \mathit{purchases}(a,i), \mathit{supplied\text{-}by}(i,s), \mathit{foreign\text{-}supplier}(s)\ ]$$

The quantifiers can now be omitted from this rule without ambiguity. In the formal semantics, ILOG$^{(\neg)}$ programs are first replaced by their Skolemizations, and then evaluated as in (stratified) logic programming. *Skolem terms* (i.e., terms in which a Skolem functor occurs) which are present in the output correspond to new OIDs. The final step of the ILOG semantics is the replacement of these Skolem terms by (non-deterministically chosen) OIDs. Intuitively, an OID is created whenever a ground Skolem term is included into some relation.

The final step of the ILOG semantics, replacing Skolem terms by OIDs, is included for two reasons: (i) to highlight the close parallel between ILOG and languages which support explicit OID creation, including IQL, OOAlgebra, and OODAPLEX; and (ii) to make the language more accessible to a wide class of potential users. The use of explicit Skolem functors yields a benefit in the succinctness of ILOG programs. In particular, the ILOG rules defining *int-aud-un*, *audit-unit*, and *agency-of* given in Example 2.1 can be abbreviated to

$$\begin{aligned} \mathit{audit\text{-}unit}(f_{a\text{-}u}(a,s)) &\leftarrow \mathit{purchases}(a,i), \\ &\qquad \mathit{supplied\text{-}by}(i,s) \\ \mathit{agency\text{-}of}(f_{a\text{-}u}(a,s),a) &\leftarrow \mathit{audit\text{-}unit}(f_{a\text{-}u}(a,s)) \end{aligned}$$

In ILOG the relationship between created OIDs and their witnesses is held in intermediate relations; if explicit Skolem terms are permitted then the OID and its witnesses can be bundled into a single term.

3 Preliminaries

We assume general familiarity with the relational model and query languages [Ull87] including *conjunctive queries* [CM77]; (stratified) datalog and logic programming [Llo87, Apt88], and semantic database models [HK87]. In this section we establish notation for these areas; due to space limitations exposition is terse.

Relational preliminaries: We assume an infinite set $\mathbf{R}$ of *relation names*. Each $R \in \mathbf{R}$ has an associated *arity* $\alpha(R) \geq 1$. A *relational (database) schema* is a set $S = \{R_1, \ldots, R_n\}$ of distinct relation names. We assume an infinite set $\mathbf{Dom}$ a universe of *domain elements*. For simplicity of presentation we do not at this point include a mechanism for assigning types to different columns of a relation instance. In the conventional approach, a (*relation*) *instance* of relation name R is a finite subset of $\mathbf{Dom}^{\alpha(R)}$, and a (*database*) *instance* of a relational schema $D = \{R_1, \ldots, R_n\}$ is a function I with domain D, where $I(R_j)$ is an instance of R_j for each $j \in [1..n]$. (We shall modify this approach in Section 4 below.) A *functional dependency* (*FD*) is a syntactic expression of the form $R : X \rightarrow Y$, where R is a relation name and $X, Y \subseteq \{1, \ldots, \alpha(R)\}$. An *inclusion dependency* (*IND*) [CFP84] is a syntactic expression of the form $R[X] \subseteq R'[Y]$, where R and R' are relation names, X is a non-repeating sequence over $\{1, \ldots, \alpha(R)\}$, and Y is a non-repeating sequence over $\{1, \ldots, \alpha(R')\}$. We assume the notion of *satisfaction* ($\models$) of an FD or IND by a database instance is well-known. A *disjointness dependency* (*DISD*) is a syntactic expression of the form $R\#R'$, where R, R' are unary relation names. A relational instance satisfies $R\#R'$ (denoted $\models$) if R and R' are assigned disjoint relations. Finally, a *constrained schema* is a pair (D, Σ) where D is a relational schema and Σ a set of constraints over D. An *instance* of this pair is an instance of D satisfying Σ.

We assume familiarity with the (domain) relational calculus. The *conjunctive queries* are a subset of the relational calculus queries whose formulas (a) are in prenex normal form, (b) use only existential quantifiers, (c) use only the connective $\wedge$ (i.e., do not use $\vee$, $\neg$ or $\rightarrow$).

(Stratified) datalog and logic programming: We first establish notation for (stratified) datalog, and then generalize to logic programming. We assume an infinite set $\mathbf{Var}$ of variables (which is disjoint from $\mathbf{Dom}$). In the context of datalog, a *term* is a variable or an element of $\mathbf{Dom}$. We assume familiarity with the notions of *atom*, *literal*, datalog$^{(\neg)}$ *rule*, rule *head* and rule *body*. We permit equality ($=$) and inequality ($\neq$) in rule bodies unless otherwise noted. An *equality atom* (*literal*) is one involving $=$ or $\neq$; an *equality-free atom* (*literal*) is one not involving $=$ or $\neq$. A datalog$^{(\neg)}$ rule is *range-restricted* if each variable occurring in the head and each variable occurring in a negated atom (this includes $\neq$ atoms) in the body also occurs in an equality-free atom in the body. We consider only range-restricted datalog$^{\neg}$ rules in the sequel.

We assume familiarity with the notion of datalog$^{(\neg)}$ *program*. We often blur a relational instance I with its associated set of atoms. We assume familiarity with the *immediate consequence* operator T_P and the *cumulative powers* $T_P \Uparrow i$ ($0 \leq i \leq \omega$) used to define the least fixpoint semantics for datalog programs. We assume familiarity with the notion of *stratification* [ABW86, vG86], and the semantics associated with stratified datalog$^{\neg}$ programs.

For a datalog$^{(\neg)}$ program P, $sch(P)$ denotes the set of relation names occurring in P. In the spirit of [AV87] we consider datalog$^{(\neg)}$ programs in connection with *source* and *target* relations. Specifically, a datalog$^{(\neg)}$ program *with source* S and *target* T is a triple (P, S, T) where (a) P is a datalog$^{(\neg)}$ program, (b) S and T are disjoint sets of relation names, and (c) no relation name in S occurs in the head of a rule of P. We do not insist that all members of S or T occur in P. We use P to denote (P, S, T) if S and T are specified by the context. Given (P, S, T), P is viewed as a mapping from instances over S to instances over T in the usual fashion. For an instance I of S, $P\langle I\rangle$ denotes the resulting instance of T.

[2] We use 'functor' instead of 'function' because, following logic programming, syntactically distinct terms will never be equated.

A fundamental premise of this paper is that OID creation can be achieved in a datalog context through the use of Skolem functors. As a result, we shall use conventional logic programming in addition to datalog. To maintain consistency with the relational point of view, we use the term 'relation name' for 'predicate'. We include a set **F** of *functor (symbols)*, where each $f \in \mathbf{F}$ has associated *arity* $\alpha(f)$. We assume familiarity with the generalization to logic programming of the concepts given above for datalog. We sometimes speak of a logic$^{(\neg)}$ program P with source S and target T, and view it as a function from "instances" of S to "instances" of T, where an "instance" might include both domain elements and terms built from functors and domain elements. In this paper, the instances of S will be instances in the conventional sense, but the instances of T may include non-atomic terms.

A semantic model: The ILOG languages are intended for use in translating one object-based schema to another, and for defining derived data in the object-based context. This paper uses a particular simple, representative semantic database model, called *IFO*$^-$. This model can be viewed as a subset of IFO [AH87] or GSM [HK87], or as a generalization of the ER [Che76] or Functional Data [Shi81] models. In particular, it supports abstract and value *object sets* (or *entity sets*), single and multi-valued *attributes* (also known as *data functions*), *aggregation* (i.e., *tuples*), and *ISA* relationships (more specifically, *specialization* relationships in the sense of [AH87, HK87]). (IFO$^-$ supports everything of the IFO model except for: the set construct; nested aggregation constructs; nested attributes; and generalization.) Due to space limitations, we present only the following abridged definition of IFO$^-$ schemas; essentially all salient features of this model are illustrated in the schemas of Figure 1.

Definition: An *IFO*$^-$ *schema* is a directed graph $S = (V, E)$ where

1. $V = V_{val} \uplus V_{abs} \uplus V_{sub} \uplus V_{agg}$ (*value* vertices, *abstract* vertices, *subtype* vertices, and *aggregation* vertices, respectively;
2. $E = E_{s\text{-}att} \uplus E_{m\text{-}att} \uplus E_{ISA} \uplus E_{comp}$ (*single-valued attribute* edges, *multi-valued attribute* edges, *ISA* (or *specialization*) edges, and *component* edges of aggregation vertices, respectively); and
3. various natural conditions concerning ISA relationships and aggregation are satisfied (e.g., see [HK87]).

Relative to this paper, an important feature of the semantic model used here is that there is a natural, direct simulation of this model by the relational model (see Section 4). The ILOG languages presented in this paper can be used with any object-based model that has this property.

4 Incorporating OIDs into semantic models

This section briefly explores differences between the original semantic models and subsequent data models which incorporate object identity as an explicit construct, and then extends the IFO$^-$ model to incorporate OIDs by developing a new definition for *instances* of a schema. This extension also leads to a variation of the relational model in the context of IFO$^-$ simulations.

Object-based models: Intuitively, *values* (e.g., integers, booleans, strings) are objects whose associated meaning is universally agreed upon, and which "carry" their own meaning. In contrast, *"abstract" objects* correspond to real or conceptual objects "in the world" for which relevant information is carried only by relationships to other objects. This distinction was originally put forth in the semantic data modeling literature [HK87]; a good recent articulation of this distinction is presented in [Bee89].

Traditionally, semantic models have assumed the existence of a (computer-representable) *surrogate* for each (real or conceptual) object "in the world". As formalized in the IFO model, there is an implicit, essentially "God-given" association between surrogates and objects in the world, which is independent of all databases instances. For example, surrogates p_1, p_2, and p_3 of type person might be associated with the persons named Joe, Mary and Sue (respectively). In this case, the relations $\{(p_1, \text{'Joe'}), (p_2, \text{'Mary'})\}$ and $\{(p_2, \text{'Joe'}), (p_3, \text{'Mary'})\}$ are distinct and in no way equivalent. Of course, in most implementations of semantic models the surrogates are created "on the fly", and so the association of surrogates to values (and implicitly, to objects in the world) is effectively made at the time of instance creation and/or extension.

The notion of explicit object identity first arose in Smalltalk-80 [GR83], and was discussed in the context of databases in [KC86]. The first theoretical database model to incorporate explicit object identifiers was LDM [KV84], this in turn significantly influenced the development of the model of IQL. Under this approach, OIDs are associated with objects "in the world" only in the context of a particular instance. Thus, if o_1, o_2, and o_3 are OIDs, then the relation $R_1 = \{(o_1, \text{'Joe'}), (o_2, \text{'Mary'})\}$ could be used to associate o_1 with the person named Joe and o_2 with the person named Mary. Also, the relation $R_2 = \{(o_2, \text{'Joe'}), (o_3, \text{'Mary'})\}$ could be used to associate o_2 and o_3 with Joe and Mary, respectively. Under the semantics of explicit OIDs, R_1 and R_2 are (in the absence of other data) interchangable, and thus viewed as equivalent. As a result, the formal semantics associated with explicit OIDs reflects the realities of database implementations more accurately than the formal semantics associated with IFO.

The formal definition of *instance* given now, which extends the direction taken in [AK89], reflects the fact that the only information held by OIDs is their interconnection with each other and with values. Speaking intuitively, an "instance" will be an equivalence class of "pre-instances" under the equivalence relation which permits shuffling of the OIDs used.

Notation: We assume an infinite set **O-dom** of *object identifiers* (*OIDs*), and a disjoint set **V-dom** of *values*. (V-dom is usually partitioned further, into integers, strings, etc.; we largely ignore those details in this abstract).

Definition: Let $S = (V, E)$ be an IFO$^-$ schema. A *pre-instance* of S is a function I with domain $V \cup E_{s\text{-}att} \cup E_{m\text{-}att}$

such that:[3]

1. for each $v \in V_{val}$, $I[v] =$ **V-dom**. (In practice, $I[v]$ need not be specified for $v \in V_{val}$.)
2. for each $v \in V_{abs}$, $I[v] \subseteq^{fin}$ **O-dom**; and for each $v, v' \in V_{abs}$ with $v \neq v'$, $I[v] \cap I[v'] = \emptyset$.
3. for each $p \in V_{sub}$ and $(p, q) \in E_{ISA}$, $I[p] \subseteq I[q]$.
4. for each $r \in V_{agg}$ with components $p_1, \ldots, p_n$ (in that order), $I[r] \subseteq^{fin} \Pi_{i=1}^{n} I[p_i]$.
5. for each $f \in E_{s\text{-}att}$ with source v and target v', $I[f]$ is a partial function from $I[v]$ to $I[v']$.
6. for each $f \in E_{m\text{-}att}$ with source v and target v', $I[f]$ is a total function from $I[v]$ to $\mathcal{P}^{fin}(I[v'])$.

Constraints can be added to make some of the single-valued attributes *total*.

Following [AK89] we have:

Definition: Let $S = (V, E)$ be an IFO$^-$ schema. Two pre-instances I, I' of S are *OID-equivalent* ($\sim_{OID}$) if there is a permutation σ on **O-dom** such that (extending σ to pre-instances in the natural fashion) $\sigma(I) = I'$. The OID-equivalence class of a pre-instance I is denoted by $[I]$.

Finally,

Definition: Let $S = (V, E)$ be an IFO$^-$ schema. An *instance* of S is an equivalence class of pre-instances under OID-equivalence.

In general, we identify an instance $\mathbf{I}$ by $[I]$ where I is some pre-instance in $\mathbf{I}$. Mappings from instances of a schema S to instances of a schema T are typically specified in terms of mappings from pre-instances over S to pre-instances over T; in this case it must be verified that the specification is *well-defined*, i.e., that it is *independent* of the representative pre-instance used. (Although using a different formalism, this sentiment is included in the definition of 'db-transformation' in [AK89].)

Relational simulations: The syntax of ILOG$^{(\neg)}$ exploits the natural correspondence between object-based schemas (in which sets are not "first-class citizens") with relational schemas. Briefly, the relational simulation (R_S, Σ_S) of an IFO$^-$ schema S is a constrained schema, where R_S associates a unary relation to each abstract and subtype node; an n-ary relation to each aggregation node of width n; and relations of appropriate arity to each single- and multi-valued attribute edge. Functional dependencies are included in Σ_S for the single-valued attribute edges; inclusion dependencies are included for each ISA edge, to restrict the ranges of attribute and component edges, and to enforce totalness constraints if they are present; and disjointness dependencies are included to ensure the separation of sets assigned to abstract types. (This is similar to the relational simulation of IRIS schemas described in [LV87].)

In this context of relational simulations of object-based schemas, we again define non-standard notions of *pre-instance* and *instance* for (constrained) relational schemas. We again use the disjoint sets **O-dom** and **V-dom**. A *pre-instance* of schema D (constrained schema (D, Σ)) is an assignment of a relation (over **O-dom** $\cup$ **V-dom**) to each relation name in D, (which satisfies all of the dependencies in Σ). An *instance* is an equivalence class of pre-instances under OID-equivalence. (Thus, the conventional notion of 'instance' is termed 'pre-instance' in this context.)

[3] $X \subseteq^{fin} Y$ denotes that X is a finite subset of Y. $\mathcal{P}^{fin}(X)$ denotes the family of all finite subsets of X.

5 Syntax and semantics of ILOG languages

In this section, we will present a formal definition of the syntax of the whole spectrum of ILOG variants ranging from the most restricted class, nonrecursive ILOG, to the most general class, stratified ILOG$^\neg$; give a semantics of ILOG$^{(\neg)}$ using Skolem functors essentially in the spirit of O-logic, as refined by 'O-logic revisited' and C-logic; and at the end of the section briefly introduce ILOG$^\pm$[HWW90], an implemented variant of ILOG.

5.1 ILOG$^\neg$: syntax

In this section, we first give the syntax of ILOG$^\neg$; this subsumes the syntax of all the languages of our interest. Then, we will give the subclasses of ILOG$^\neg$ along the two orthogonal dimensions: negation and recursion. Along the negation dimension, we will present two positions: no negation and stratified negation. As for the recursion dimension, we will consider three positions: nonrecursive, weakly recursive and fully recursive. Hence, the combination of these subclasses yields six sublanguages as a whole (see Figure 2.)

Definition: (*ILOG*$^\neg$) An *invention atom* is an expression of the form $R(*, t_1, \ldots, t_m)$ where R is a relation name with $\alpha(R) = m + 1$ ($m \geq 0$), $*$ is a special symbol called the *invention symbol* and the t_i ($1 \leq i \leq m$) are terms. In ILOG$^\neg$, a *rule* is an expression of the form $A \leftarrow L_1, \ldots, L_n$ where the *body* $L_1, \ldots, L_n$ ($n \geq 1$) is a sequence (viewed as a conjunction) of (positive or negative) literals (possibly including equality literals), and the *head* A is an equality-free atom (possibly involving invention). A rule is a *non-invention* (*invention*, resp.) rule if the head is a non-invention atom (invention atom, resp.). A *program* is a finite set of rules with a restriction that no relation name can appear in the head of both a non-invention rule and an invention rule.[4] A relation name appearing in the head of an invention rule is an *invention relation name*.

The following notion of *range-restricted* ILOG$^\neg$ rule is a straightforward extension of the one for datalog$^\neg$: An ILOG$^\neg$ rule is *range-restricted* if each variable occurring in the head and each variable occurring in a negated atom (this includes $\neq$ literals) in the body also occurs in a positive equality-free atom in the body. We claim the majority of ILOG$^\neg$ programs which appear in practice are range-restricted. Therefore, hereafter throughout the paper, we consider only range-restricted ILOG$^\neg$ rules.

An ILOG$^\neg$ program P *with source* S and *target* T, (P, S, T), can also be defined in a manner analogous to the case of datalog$^\neg$.

[4] It is easy to verify that this restriction does not cause any loss of expressive power.

We now define the sublanguages of ILOG$^\neg$. The first dimension to be explored is that of negation. The two positions along this dimension, no negation and stratified negation, correspond exactly to datalog and stratified datalog$^\neg$, respectively. Thus, formal definitions of these sublanguages are omitted here. Since stratified ILOG$^\neg$ is the most general class in this dimension, we use ILOG$^\neg$ to mean stratified ILOG$^\neg$ in the sequel.

We now consider the second dimension: recursion. *Fully recursive* ILOG$^{(\neg)}$ is exactly the same as ILOG$^{(\neg)}$; hence we usually omit the term *fully recursive*. The definition of *non-recursive* ILOG$^{(\neg)}$ is analogous to that of datalog$^{(\neg)}$, and is omitted. Intuitively, an ILOG$^{(\neg)}$ program is *weakly recursive* if it is free of recursions through OID creation. We will use *nrecILOG*$^{(\neg)}$ and *wrecILOG*$^{(\neg)}$ to denote non-recursive and weakly recursive ILOG$^{(\neg)}$, respectively.

Definition: (*weakly recursive ILOG$^\neg$ programs*) Given an ILOG$^\neg$ program P, let P' an equivalent program resulting from the removal of equality atoms from P (in some predetermined canonical fashion; '$\neq$' is permitted). Consider pairs (R, i) for each relation name R in P and each position $i \in [1..\alpha(R)]$. We say that (R,i) **-derives* (Q,j) in P, denoted $(R,i) \rightsquigarrow (Q,j)$, as follows:

1. $(R,i) \rightsquigarrow (Q,j)$ if there is a rule in P' of the form $Q(s_1,\ldots,s_j,\ldots,s_n) \leftarrow \ldots, R(t_1,\ldots,t_i,\ldots,t_m),\ldots$ where $t_i = s_j$ is a variable;
2. $(R,i) \rightsquigarrow (Q,j+1)$ if there is a rule in P' of the form $Q(*, s_1,\ldots,s_j,\ldots,s_n) \leftarrow \ldots, R(t_1,\ldots,t_i,\ldots,t_m),\ldots$ where $t_i = s_j$ is a variable; and
3. $(Q,j+1) \rightsquigarrow (Q,1)$ if there is a rule in P' of the form $Q(*, s_1,\ldots,s_j,\ldots,s_n) \leftarrow \ldots$ where s_j is a variable.

We define $\overset{+}{\rightsquigarrow}$ to be the transitive closure (*not* the reflexive transitive closure) of $\rightsquigarrow$. An ILOG$^{(\neg)}$ program P is *weakly recursive* if it does not contain any invention relation name R such that $(R,1) \overset{+}{\rightsquigarrow} (R,1)$.

It is easy to see that every non-recursive ILOG$^\neg$ is weakly recursive.

5.2 Declarative semantics of ILOG programs using Skolem functors

In this subsection we describe the declarative semantics associated with ILOG$^{(\neg)}$ programs. We distinguish two contexts: schema *translation*, where there is no relationship between the OIDs of the source instance and the OIDs of the target instance; and schema *augmentation*, where the target (pre-)instance is permitted to refer directly to OIDs of the source (pre-)instance. We begin by focusing exclusively on schema translation, and then modify the semantics for the case of augmentation.

Suppose now that (P,S,T) is an ILOG$^{(\neg)}$ program, considered in the context of schema translation, and where S and T are relational schemas. In principle, the semantics of P defined here will be a function from instances of S to instances of T. This function might be partial because P may call for the creation of an infinite instance (or in other words, the computation associated with P may not terminate).

The function defined by P will be specified in terms of its behavior on pre-instances of S. (This is compatible with most implementations, which indeed store a single pre-instance of an object-based database.) This function is obtained through a three-step process, briefly outlined now:

1. construct the 'Skolemization' $Skol(P)$ of P by introducing Skolem functors;
2. for an instance $\mathbf{I} = [I]$ of S, compute[5] $Skol(P)\langle I\rangle$ using conventional (or stratified) logic programming, to obtain a 'Skolemized' pre-instance, that is, a pre-instance in which Skolem functors may occur;
3. compute[5] the set $MkInstTr(Skol(P)\langle I\rangle)$ of all pre-instances J obtained from $Skol(P)\langle I\rangle$ by mapping each non-value term of $Skol(P)\langle I\rangle$ to a distinct (non-deterministically chosen) OID. As we shall see, this set of pre-instances is an instance of T, and is the value assigned to $P\langle \mathbf{I}\rangle$.

If in step (2) above $Skol(P)\langle I\rangle$ yields an infinite set, then $P\langle \mathbf{I}\rangle$ is *undefined.* (If constraints are associated with T, e.g., if it is the relational simulation of an IFO$^-$ schema, then the output $P\langle \mathbf{I}\rangle$ may violate some of those constraints. We defer consideration of this issue until Section 8; until then we assume that the target relational schema has no associated constraints.)

We now consider step (1) in more detail. Following the intuition described in Section 2, we have:

Definition: Let P be an ILOG$^{(\neg)}$ program. The *Skolemization* of P, denoted $Skol(P)$ is obtained by the following transformation from P

(a) For each n-ary relation name R ($n \geq 1$) which appears in the head of an invention rule in P, introduce a new $(n-1)$-ary functor f_R (the *Skolem functor* of R).

(b) Replace each head of an invention rule in P having the form $R(*, x_1,\ldots,x_{n-1})$ by $R(f_R(x_1,\ldots,x_{n-1}), x_1,\ldots,x_{n-1})$.

Example 5.1: We show an ILOG$^\neg$ program P, and $Skol(P)$ below it.

$$\begin{array}{ll}
R(*,x,y) & \leftarrow U(x,y) \\
R(*,x,x) & \leftarrow V(x,y), \neg W(y) \\
V(*,x) & \leftarrow W(x), \neg U(x,y) \\
\\
R(f_R(x,y),x,y) & \leftarrow U(x,y) \\
R(f_R(x,x),x,x) & \leftarrow V(x,y), \neg W(y) \\
V(f_V(x),x) & \leftarrow W(x), \neg U(x,y)
\end{array}$$

□

Note that if P is in ILOG$^\neg$, then $Skol(P)$ is stratified. To describe step (2), we informally introduce: A *Skolem term* is a term in which at least one Skolem functor occurs. A *Skolemized pre-instance* of a relational schema S is like a pre-instance, except that Skolem terms may occur (in addition to values and OIDs). For an instance $\mathbf{I} = [I]$ of S, $Skol(P)\langle I\rangle$ is a Skolemized pre-instance over T.

Step (3) of the process is intended to yield the instance of T which is identified by $Skol(P)\langle I\rangle$. In particular, all OIDs

[5]We use the term 'compute' for readability, but this can be specified declaratively.

and Skolem terms occurring in $Skol(P)\langle I \rangle$ are to be viewed as undistinguished OIDs – relationships of Skolem terms and their components are to be ignored (removed), and relationships of OIDs in $Skol(P)\langle I \rangle$ and I are to be ignored. This is accomplished formally by the function $MkInstTr$:

Definition: $MkInstTr$ is the mapping from Skolemized pre-instances to sets of pre-instances defined so that for each Skolemized pre-instance I, $MkInstTr(I) = \{\rho(I) \mid \rho$:**O-dom** $\cup$ (Skolem terms in I) $\rightarrow$ **O-dom** is a 1-1 mapping$\}$.

It is straightforward to verify that:

Proposition 5.2: Let (P, S, T) be an ILOG$^{(\neg)}$ program.

(a) If I is a pre-instance of S then $Skol(P)\langle I \rangle$ is infinite or $MkInstTr(Skol(P)\langle I \rangle)$ is an instance of T; and

(b) If $I \sim_{OID} I'$ then both $Skol(P)\langle I \rangle$ and $Skol(P)\langle I' \rangle$ are infinite, or $MkInstTr(Skol(P)\langle I \rangle) = MkInstTr(Skol(P)\langle I' \rangle)$. Thus, the mapping $\mathbf{I} = [I] \mapsto MkInstTr(Skol(P)\langle I \rangle)$ is well-defined.

In view of this proposition, we can safely define the semantics of (P, S, T) (in the context of schema translation) by $P\langle \mathbf{I} \rangle = MkInstTr(Skol(P)\langle I \rangle)$ if $Skol(P)\langle I \rangle$ is finite, and undefined otherwise, where I is some (any) pre-instance in $\mathbf{I}$.

Remark 5.3: In the formal semantics just established for ILOG$^{(\neg)}$ in the context of schema translation, OIDs in the target pre-instance "lose" their association with the source pre-instance. In practice, it may be appropriate to discourage programs for which OIDs (outside of Skolem terms) in the target pre-instance might overlap with the source pre-instance OIDs, because the true semantics may not correspond to the programmer's intended semantics. On the other hand, permitting this kind of overlap can simplify ILOG$^{(\neg)}$ programs. For example, a program equivalent to that of Example 2.2 can be obtained by replacing the rules there for *int-ra-agency*, *ra-agency*, and *audits-of* by

$$\begin{array}{lcl} ra\text{-}agency(a) & \leftarrow & agency(a), int\text{-}aud\text{-}un(u,a,s), \\ & & \quad required\text{-}audit(u) \\ audits\text{-}of(a,u) & \leftarrow & agency(a), int\text{-}aud\text{-}un(u,a,s), \\ & & \quad required\text{-}audit(u) \end{array}$$

□

We now turn to the use of ILOG$^{(\neg)}$ for schema augmentation, i.e., for defining derived data. Again, suppose that (P, S, T) is an ILOG$^{(\neg)}$ program, where S and T are relational schemas. Suppose further that I is a pre-instance of S which is currently stored by some implementation. The result of applying P to I may include OIDs from I, and this association *should not be destroyed* as it was in the context of schema translation. For this reason, the third step listed above for the semantics of schema translation is modified in the context of schema augmentation, and a fourth is added:

3′. compute[5] the set $MkInstAug(I, Skol(P)\langle I \rangle)$ of all pre-instances J obtained from $Skol(P)\langle I \rangle$ by leaving each OID in I fixed, and mapping each non-atomic term of $Skol(P)\langle I \rangle$ to a distinct, new (non-deterministically chosen) OID.

4. Nondeterministically choose $J \in MkInstAug(I, Skol(P)\langle I \rangle)$ to serve as the output of the computation.

To provide more detail, we first (informally) establish the following notation: If S and T are disjoint relational schemas, then $S \oplus T$ denotes their union. If I and J are pre-instances of S and T, respectively, then $I \oplus J$ denotes the "union" of I and J, a pre-instance over $S \oplus T$. Also, $Obj(I)$ denotes the set of OIDs which occur in I.

Suppose now that a pre-instance I is fixed. As before, the semantics of P on I is given in three steps. The first two are as before, yielding the computation of $Skol(P)\langle I \rangle$, which is (infinite or) a Skolemized pre-instance of T. This is transformed into a set of pre-instances using the function:

Definition: $MkInstAug$ maps pairs (I, J), where I is a pre-instance of S and J is a Skolemized pre-instance of T, into sets of pre-instances of T as follows: $MkInstAug(I, J) = \{\rho(J) \mid \rho$:**O-dom** $\cup$ (Skolem terms in J) $\rightarrow$ **O-dom** is 1-1 and leaves $Obj(I)$ fixed $\}$.

It is straightforward to verify that:

Proposition 5.4: Let (P, S, T) be an ILOG$^{(\neg)}$ program.

(a) If I is a pre-instance of S then $Skol(P)\langle I \rangle$ is infinite or $MkInstAug(I, Skol(P)\langle I \rangle)$ is an equivalence class of pre-instances under the equivalence relation $\sim_{OID[Obj(I)]}$, where for each $X \subseteq$ **O-dom**, $J \sim_{OID[X]} J'$ iff there is some permutation σ on **O-dom** which leaves X fixed such that $J' = \sigma(J)$.

(b) Let $\mathbf{I}$ be an instance of S. Then $Skol(P)\langle I \rangle$ is infinite for some $I \in \mathbf{I}$ iff $Skol(P)\langle I \rangle$ is infinite for each $I \in \mathbf{I}$. Furthermore, if $Skol(P)\langle I \rangle$ is finite, then the set $\{I \oplus J \mid I \in \mathbf{I}$ and $J \in MkInstAug(I, Skol(P)\langle I \rangle)\}$ is an instance of $S \oplus T$.

In view of this proposition, we can safely define the semantics P_{aug} of (P, S, T) in the context of schema augmentation so that for each pre-instance I of S, $P_{aug}\langle I \rangle = MkInstAug(I, Skol(P)\langle I \rangle)$ if $Skol(P)\langle I \rangle$ is finite, and undefined otherwise. In practice, application of P to a pre-instance I in the context of schema augmentation will yield the pre-instance $I \oplus J$, where J is a non-deterministically chosen member of $P_{aug}\langle I \rangle$.

Remark 5.5: A problem arises in the object-based context if derived data is to be (re-)computed as needed rather than computed once and stored. Specifically, under the semantics just established, the OIDs used for newly created objects in the derived data may be different for each re-computation. To overcome this problem in practice, a regime can be established which assigns OIDs to non-ground Skolem terms in a systematic manner. □

5.3 ILOG$^{\pm}$: an implemented ILOG language

The research reported in this paper occurred in parallel with the development and prototype implementation of another ILOG variant, called ILOG$^{\pm}$ [HWW90]. This practically motivated language will be used in connection with a larger project supporting database sharing [WHW89].

For the purposes of this paper, we view ILOG$^{\pm}$ as nrecILOG augmented with the ability to use the full relational calculus on source instances. (ILOG$^{\pm}$ as described in [HWW90] is actually stronger, because it also has the ability to use external functions defined by LISP subroutines. This is because the ILOG$^{\pm}$ prototype is built on the database programming langauge AP5 [Coh86, Coh89], which is an extension of Common Lisp. There are also some essentially syntactic differences between the ILOG$^{\pm}$ described here and described in [HWW90].)

Formally, an ILOG$^{\pm}$ *rule* from relational source schema S to relational target schema T has the form $A \leftarrow L_1, \ldots, L_n$ where A is an atom (possibly with invention), and where each L_i is either a positive atom or a relational calculus formula over S with 0 or more free variables. An ILOG$^{\pm}$ *program* is a set of ILOG$^{\pm}$ rules which is non-recursive. The semantics of ILOG$^{\pm}$ programs is given in the natural, bottom-up fashion. As shown in Theorem 7.3, ILOG$^{\pm}$ has expressive power equivalent to that of nrecILOG$^{\neg}$.

6 Explicit OID creation semantics of ILOG

In this section, we will give the definition of an explicit OID creation semantics for ILOG programs essentially in the spirit of detTL, detDL and IQL (a comparison of this semantics to IQL semantics is presented at the end of Section 7), and show that this semantics is equivalent to the Skolem functor semantics defined in the previous section. We focus here on the case of schema translation; analogous results hold for schema augmentation.

Definition: (*witness*) Let P be an ILOG program, and I a pre-instance of $sch(P)$. Also, let R be an n-ary invention relation name in P. A tuple $\vec{a} = (a_1, \ldots, a_{n-1})$ is a *witness* (i.e. witnessing the need for a new OID) for R under P on I if

1. there exists an invention rule r of the form $R(*, x_1, \ldots, x_{n-1}) \leftarrow L_1, \ldots, L_m$ in P;
2. there exists a ground substitution θ such that $I \models L_j\theta$ for $j \in [1, \ldots, m]$ and $\theta(x_i) = a_i$ for $i \in [1, \ldots, n-1]$; and
3. there is no way to extend θ (by assigning a member of **O-dom** to the $*$-symbol) such that $I \models R(*, x_1, \ldots, x_{n-1})\theta$.

Definition: (*semantics of ILOG programs using direct OID semantics*) For an ILOG program P, $\hat{T}_P$ is a binary relation on pre-instances of $sch(P)$. The pair of pre-instances (I, J) is in $\hat{T}_P$ if: $J = I \cup \{A\theta \mid A \leftarrow L_1, \ldots, L_m$ is in P; A is not an invention atom; and θ is a ground substitution such that $I \models L_i\theta$ for $i \in [1, \ldots, m]\} \cup \{R(o_{R,\vec{a}}, a_1, \ldots, a_{n-1}) \mid R$ is an invention relation name; $\vec{a} = (a_1, \ldots, a_{n-1})$ is a witness for R under P on I; and $o_{R,\vec{a}}$ is a new, distinct OID for each invention relation name R and witness tuple $\vec{a}$ for R under P on I $\}$ Given a program P, the *fixpoint operator* $\hat{T}_P \uparrow \omega$ is a binary relation on pre-instances defined as follows: The pair (I, J) of pre-instances of $sch(P)$ is in $\hat{T}_P \uparrow \omega$ if: (1) there exists a sequence $I = I_0, I_1, \ldots$ of instances of $sch(P)$ such that for each i $(0 \leq i < \omega)$, the pair (I_i, I_{i+1}) is in $\hat{T}_P$; (2) $J = \cup_{i<\omega} I_i$; and (3) J is finite.

The explicit OID creation semantics of an ILOG program (P, S, T) is defined as a mapping $\hat{P}$ from pre-instances of S to sets of pre-instances of T such that for a given pre-instance I of S, $\hat{P}\langle I \rangle$ is undefined if $\{J \mid (I, J) \in \hat{T}_P \uparrow \omega\}$ is empty, and is[6] $\{J[T] \mid (I, J) \in \hat{T}_P \uparrow \omega\}$ otherwise. (Intuitively, $\hat{P}\langle I \rangle$ is undefined if some (any) "computation" of $\hat{T}_P$ on I is nonterminating.)

This semantics is extended to ILOG$^{\neg}$ in the natural fashion. It is now straightforward to show:

Proposition 6.1: Let (P, S, T) be an *ILOG*$^{(\neg)}$ program with source and target schemas.

(a) If I is a pre-instance of S then $\hat{P}\langle I \rangle$ is undefined, or is an instance of T; and

(b) For any two pre-instances I_1 and I_2 of S, if $I_1 \sim_{OID} I_2$, then both $\hat{P}\langle I_1 \rangle$ and $\hat{P}\langle I_2 \rangle$ are undefined, or $\hat{P}\langle I_1 \rangle = \hat{P}\langle I_2 \rangle$.

Proposition 6.1, as in the case of Skolem functor semantics in the previous section, permits us to view $\hat{P}$ as a mapping from instances to instances: For an instance $\mathbf{I}$ of S, $\hat{P}\langle \mathbf{I} \rangle$ is the instance on T such that $P\langle \mathbf{I} \rangle = [J]$ where $J \in \hat{P}\langle I \rangle$ and I is some (any) pre-instance of S such that $\mathbf{I} = [I]$ if $\hat{P}\langle I \rangle$ is defined; $\hat{P}\langle \mathbf{I} \rangle$ is undefined otherwise. The following theorem establishes, in the context of schema translation, the equivalence of the two semantics of ILOG$^{(\neg)}$, namely Skolem functor semantics defined in the previous section and the explict OID creation semantics defined above.

Theorem 6.2: Let (P, S, T) be an ILOG$^{(\neg)}$ program with source and target schemas. For any instance $\mathbf{I}$ of S, both $P\langle \mathbf{I} \rangle$ and $\hat{P}\langle \mathbf{I} \rangle$ are undefined, or $P\langle \mathbf{I} \rangle = \hat{P}\langle \mathbf{I} \rangle$.

An analogous result can be obtained in the context of schema augmentation.

7 Expressive power

This section establishes a number of results concerning the relative expressive power and complexity of the ILOG languages. The weakest of the languages is nrecILOG, which is closely related to the conjunctive queries, but more succinct. The strongest of the languages is ILOG$^{\neg}$ – similar to IQL this langauge can express all computable database translations up to "copy removal". A key element of the development is a result specifying normal forms for nrecILOG and wrecILOG$^{\neg}$ programs. The section also analyzes some salient differences between ILOG$^{(\neg)}$ and IQL, and briefly compares some more practical schema translation languages with ILOG$^{(\neg)}$.

We begin by presenting normal forms for nrecILOG and wrecILOG$^{\neg}$ programs. Intuitively, the proposition states these programs can be rewritten so that (i) there is no "cascading" of OID creation, and (ii) OID creation can be "delayed" until the last stage of the "computation". To define

[6]If schema S_2 is contained in schema S_1 and K is a pre-instance of S_1, then $K[S_2]$ is the restriction of K to S_2.

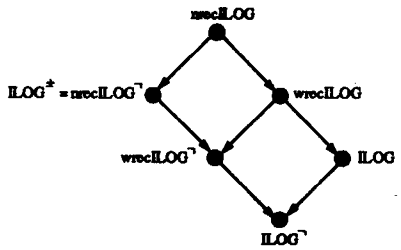


Figure 2: Relative expressivity of ILOG languages

the notion of cascading formally, given an ILOG$^{(\neg)}$ program P, write $R \Rightarrow R'$ if there is a rule in P with head relation name R' and relation name R occurring positively in the body; and let $\overset{+}{\Rightarrow}$ denote the transitive closure of $\Rightarrow$. Then *P has cascading of OID creation* if there is a pair R, R' of invention relation names such that $R \overset{+}{\Rightarrow} R'$. We now have:

Proposition 7.1: (Normal Forms)

1. If P is a nrecILOG program then:
   (a) there is a nonrecursive logic program L_P which is equivalent to $Skol(P)$ such that the body of each rule contains only source relation names.
   (b) there is a nrecILOG program P' equivalent to P which does not have cascading of OID creation.
2. If P is a wrecILOG¬ program then:
   (a) there is a logic¬ program L_P equivalent to $Skol(P)$ such that if a functor occurs in a literal A of L_P with relation name R, then R does not occur in any rule body; and for at least one stratification of L_P, all functors occur in the heads of rules in the last stratum of L_P.
   (b) there is a wrecILOG¬ program P' equivalent to P which does not have cascading of OID creation; and such that for at least one stratification of P', all invention relation names occur in the last stratum of P'.

Proof: (Sketch) The proofs rely on the fact that in the absence of $*$-recursion, an ILOG$^{(\neg)}$ program can simulate a Skolem term $f(a_1, \ldots, a_n)$ with the tuple $(a_1, \ldots, a_n)$. (This can be generalized to Skolem terms of arbitrary fixed depths.) Using this, the argument shows that cascading of OID creation can be eliminated, and that OID creation can be "pushed" through the strata of a wrecILOG¬ program. □

The normal forms of the above proposition can be exponential in the size of the original ILOG$^{(\neg)}$ program. The normal form proposition and results of [Var82] imply:

Proposition 7.2: The data complexity of wrecILOG is NLOGSPACE, and of wrecILOG¬ is between NLOGSPACE and PTIME. The output $Skol(P)(I)$ of a wrecILOG¬ program on input pre-instance I can be computed in PTIME (in terms of the size of I).

We now present the main theoretical result of the section. We say that an ILOG language L is (*properly*) *subsumed* by language L' (denoted using $\sqsubset$, $\sqsubseteq$) if each mapping realized by a program in L is also realized by a program in L'. L and L' are *equivalent* (denoted $\equiv$) if $L \sqsubseteq L'$ and $L' \sqsubseteq L$. The results of the following theorem are summarized in Figure 2.

Theorem 7.3: If (L, L') is an edge in the graph of Figure 2 then $L \sqsubset L'$; and if L, L' are two languages which are not comparable in the figure, then their expressive powers are incomparable. Finally, ILOG$^{\pm} \equiv$ nrecILOG¬.

We now have a sequence of results for ILOG¬ showing that it comes "close" to specifying all computable database translations. These results are similar to results for IQL presented in [AK89]. Essentially in the spirit of that paper, we have[7]:

Definition: Let S and T be IFO^{-} schemas. A mapping from instances of S to instances of T is a *database translation* if it is (a) Turing computable (under some fixed encoding scheme) and (b) generic (in the sense of [CH80, Hul86]).

We now informally introduce a notion which is essentially the same as 'schema with copies' from [AK89], but relativized to IFO^{-}. Intuitively, a *copy-version* of an instance $\mathbf{I}$ of S is an instance $\mathbf{J} = [J]$ over a new schema S^{+}, where J contains one or more OID-equivalent copies of pre-instances of $\mathbf{I}$. The schema S^{+} is augmented by a single new abstract vertex v_{index}, which is used to distinguish the different copies of a copy version. (The formal definition is omitted for space reasons.) Again in the spirit of [AK89], we have:

Definition: Let F be a database translation from S to T. Then G from S to T^{+} *computes F modulo copy removal* if for each instance $\mathbf{I}$ of S, (i) $G(\mathbf{I})$ is undefined if $F(\mathbf{I})$ is undefined, and (ii) $G(\mathbf{I})$ is a copy-version of $F(\mathbf{I})$ if $F(\mathbf{I})$ is defined.

We can now state the following analog of a result for IQL; a key element of the proof is demonstrating that ILOG¬ has the power to create an arbitrary number of OIDs while avoiding nontermination.

Theorem 7.4: ILOG¬ can express all computable database translations modulo copy removal.

As with IQL, the above theorem easily implies that ILOG¬ can specify all database translations whose range is contained in any of the following sets: (a) {YES, NO}; (b) instances in which no OIDs occur; (c) instances in which no values occur. (Another recent logic-based langauge which can express all computable queries is IDLOG [She90]. This language does not involve OIDs, but the presence of integers and addition enables the "creation" of an arbitrary amount of "work-space".)

It was recently shown that in the absence of copy-removal, neither IQL nor ILOG¬ express all computable queries [AK90]. Furthermore, as shown in the next example, in the absence of copy-removal ILOG¬ is strictly

[7]The definition here appears to be simpler than that of [AK89]; some intricacy here is hidden in the definition of 'instance' as equivalence class of pre-instances.

weaker than IQL, even when restricted to input and output not involving sets. This difference in the power of the two languages can be traced to the fact that IQL supports a "grouping" construct (as in LDL [NT89]) while ILOG$^\neg$ does not.

Example 7.5: Consider the IFO$^-$ schema $S = (V, E)$ where there is one abstract vertex v_a, one value vertex v_v, and one multi-valued attribute edge f from v_a to v_v. For a pre-instance I of S define the equivalence relation $\sim_I$ on $I[v_a]$ by: $o \sim_I o'$ iff $I[f](o) = I[f](o')$. Let F be the computable database translation from S to S which has the following behavior: $F([I]) = [J]$ where for each equivalence class $\mathbf{o} = [o]_{\sim_I}$ there is exactly one element $p_{\mathbf{O}} \in J[v_a]$, and $J[f](p_{\mathbf{O}}) = I[f](o)$. Intuitively, F collapses OIDs of $[I]$ which are equivalent under $\sim_I$.

To show that there is no ILOG$^\neg$ program that computes F, suppose for the sake of contradiction that P does. Let a, b, c, d be values which do not occur in P. Consider now the input pre-instances I_n of S, described here by enumerating the pairs in $R_f[I]$ (where a, b, c, d are values, and x_i, y_i are OIDs), where: $I_n = \{(x_i, a), (x_i, b), (y_i, c), (y_i, d) \mid i \in [1..n]\}$. Using the fact that all OIDs created by P on inputs $[I_n]$ have a representation using Skolem functors, it can be shown that P does not compute F.

In contrast, IQL can compute F through the use of the grouping construct. In particular, IQL can use grouping to form a collection of all sets $I[f](o)$ for $o \in I[v_a]$. These sets can in turn be used to create new OIDs, one for each equivalence class OIDs in $I[v_a]$. □

We now turn to comparing the semantics for OID creation in ILOG$^{(\neg)}$ with those of IQL. Both languages provide an explicit mechanism to create new objects, based on the existence of a tuple of already existing values and objects which satisfy certain conditions. IQL permits object creation in different and possibly multiple columns of a relation name, whereas ILOG permits object creation only in the first column of selected relation names. This difference is largely cosmetic.

The semantics of an IQL program is given through an iterative procedure, namely "firing" the set of rules of the program in parallel. This procedure is closely related to the use of the cumulative powers of the immediate consequence operator to construct least fixpoints, but with an important difference. In both ILOG$^{(\neg)}$ and IQL, if an object is created for a given rule and tuple at some point, then new objects will not be created for that rule and tuple in subsequent iterations. In the first iteration for which a tuple newly satisfies some rule for a relation name R, ILOG$^{(\neg)}$ will create exactly one new OID for that R and that tuple. In contrast, IQL may create more than one new OID for R and the tuple - it creates a new OID for each *rule* defining R for which this tuple is satisified. This feature of IQL semantics and its implications are explored in the following example.

Example 7.6: We introduce a variant of the ILOG semantics, called ILOG$'$ semantics, which corresponds to a fundamental aspect of the IQL semantics. We show that without negation, ILOG$'$ semantics is fundamentally *non-monotonic* and strictly richer than ILOG semantics.

Recall the explicit OID creation semantics for ILOG programs given in Section 6 above. The ILOG$'$ semantics is given in the same fashion, with the following difference: In an iteration starting with pre-instance I, a new OID is created for each rule r with head $p(*, x_1, \ldots, x_n)$ and tuple $(a_1, \ldots, a_n)$ such that (a) the body of r is satisfied by some assignment θ with $\theta(x_i) = a_i$ for $i \in [1..n]$, and (b) there is no b such that $p(b, a_1, \ldots, a_n)$ is already true in I. The difference between this and the ILOG semantics is that here, a new OID is created for each *rule* (under certain conditions), whereas in ILOG a new OID is created for each *relation name* (under certain conditions).

It is relatively straightforward to show that each ILOG program can be simulated by an ILOG$'$ program. We now exhibit an ILOG$'$ program P which cannot be simulated by an ILOG program. In this program, \$ and @ are constants, R and R' are source relations, and *new* is the target relation:

$$\begin{array}{ll}
R_2(x,z) \leftarrow R(x,z) & R_2'(x,z) \leftarrow R'(x,z) \\
R_1(x,y) \leftarrow R(x,y) & R_1'(x,y) \leftarrow R'(x,y) \\
R_2(x,z) \leftarrow R_1(x,y), R_1(y,z) & R_2'(x,z) \leftarrow R_1'(x,y), R_1'(y,z) \\
new(*) \ \leftarrow R_2(\$, @) & new(*) \ \leftarrow R_2'(\$, @)
\end{array}$$

Let $I = \{R(\$, 1), R(1, @), R'(\$, 1), R'(1, @)\}$ and $J = I \cup \{R(\$, @)\}$. Under ILOG$'$ semantics, P on I creates exactly two new OIDs (in the third iteration of parallel rule execution) while P on J creates exactly one new OID (in the second iteration). Since $I \subseteq J$, this implies that ILOG$'$ is essentially *non-monotonic*. Because ILOG semantics is rooted in logic programming, which is monotonic, no ILOG program (without negation) can simulate the behavior of P. □

We conclude this section with a brief comparison of sublanguages of ILOG$^{(\neg)}$ with four more pragmatically motivated data manipulation languages in the literature; more details can be found in [HY91]. The first is the language for "superview construction" [BM81, Mot87]: this language can be simulated in its entirety by nrecILOG. The three other langauges are: the transformation language of Multibase system [DH84], the derivation operators of the Federated approach [HM85], and OOAlgebra [Day89]. Each of these incorporates set difference, and so cannot be simulated by nrecILOG. Each also includes other features, such as arithmetic operators or multisets, which fall outside of the scope of the ILOG languages as presented here. However, the "core" of each of these three languages can be simulated by nrecILOG$^\neg$.

8 Testing validity of translations

An ILOG$^{(\neg)}$ program is *valid* if it maps instances of the source schema to instances of the target schema. The mapping may be invalid because the assigned value of a single-valued attribute in the target may not be single-valued, or because a subset or disjointness relationship required by the target schema is not satisfied. The notion of validity can also be extended to encompass totalness and other integrity constraints on the source and/or target.

In this section we study the problem of testing validity at compile time of ILOG$^{(\neg)}$ with IFO$^-$ schemas. It turns out that several validity issues (specifically, functional dependency, subset relationships, and disjointness relationships) are decidable for nrecILOG programs (possibly involving '$\neq$'). These results are also true if totalness constraints are permitted on single-valued attributes of the source and/or target schemas, provided that '$\neq$' is not permitted. (The case with both totalness constraints and '$\neq$' remains open.) On the other hand, essentially all validity issues are undecidable for nrecILOG$^\neg$ and wrecILOG.

We begin with formal definitions, for both the object-based and relational contexts.

Definition: Let S and T be IFO$^-$ schemas, possibly with constraints. A mapping F on instances of S is *valid* if for each instance $\mathbf{I}$ of S, $F(\mathbf{I})$ is an instance of T or is undefined. If P is an ILOG$^{(\neg)}$ program from S to T, then P is *valid* if for each pre-instance I of S, $P\langle I\rangle$ is a pre-instance of T or is undefined (because $Skol(P)\langle I\rangle$ is infinite).

In the spirit of [AH88], the notion of validity can essentially be recast in the context of the relational model using the notion of "implication problems":

$\Phi - \Psi$ Implication Problem for language Λ: Let Φ,Ψ be classes of relational dependencies and Λ be a relational language. Given a program (P, S, T) in language Λ, $\Sigma \subseteq \Phi$ over S and $\Gamma \subseteq \Psi$ over T, is it true that: for all I over S, $I \models \Sigma$ and $P\langle I\rangle$ defined implies $P\langle I\rangle \models \Gamma$?

By generalizing techniques and results from [Klu80] ([AH88]) we obtain:

Theorem 8.1: The following problems are undecidable for nrecILOG$^\neg$ (wrecILOG) programs:

(a) $\emptyset$-FD implication (FD-FD implication)

(b) $\emptyset$-IND implication ($\emptyset$-IND implication)

(c) $\emptyset$-disjointness implication (FD-disjointness implication)

Furthermore, for both languages, validity with respect to IFO$^-$ schemas as source and target is undecidable.

We turn now to decidability results for nrecILOG programs. The first result uses the Normal Form proposition to generalize results of [KP82].

Theorem 8.2: The FD-FD implication problem is decidable for nrecILOG (possibly with '$\neq$'). The complexity of this problem is at least PSPACE, and is bounded above by EXPTIME.

Because of the close relationship between nrecILOG programs and conjunctive queries, there is a close relationship between Φ-IND implication problems for nrecILOG and testing containment between conjunctive queries applied in the context of dependencies from Φ. Unfortunately, (FD $\cup$ IND)-IND implication is in general undecidable for conjunctive queries, since logical implication of an IND by a set of FDs and INDs is undecidable [Mit83]. Positive results can be obtained, however, if we restrict attention to nrecILOG programs whose source and target are (relational simulations of) IFO$^-$ schemas. In particular, by generalizing tableaux chasing techniques and results from [JK84] we have:

Theorem 8.3: It is decidable, given an nrecILOG program P from IFO$^-$ source S to IFO$^-$ target T, whether $P\langle\mathbf{I}\rangle$ is an instance of T for each instance $\mathbf{I}$ of S. Furthermore, this result continues to hold if some single-valued attributes of source and/or target are required to be total but '$\neq$' is not permitted. The complexity of these problems is at least PSPACE and is bounded above by EXPSPACE.

Acknowledgements

The authors are grateful to (Serge Abiteboul and Paris Kanellakis) [Surjatini Widjojo and Dave Wile] for numerous provocative discussions about (theoretical) [practical] aspects of languages for OIDs.

References

[AB86] S. Abiteboul and N. Bidoit. Nonfirst normal form relations: An algebra allowing data restructuring. *J. Comput. Syst. Sci.*, 33:361–393, 1986.

[Abi89] Serge Abiteboul. Towards a deductive object-oriented database language. In *Proc. of First Intl. Conf. on Deductive and Object-Oriented Databases*, pages 419–438, 1989.

[ABW86] K. Apt, H. Blair, and A. Walker. Toward a theory of declarative knowledge. In *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, 1986.

[AG88] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In *Advances in Database Technology - EDBT '88*. Lecture Notes in Computer Science, Springer-Verlag, 1988.

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, Dec. 1987.

[AH88] S. Abiteboul and R. Hull. Data functions, DATALOG, and negation. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 143–153, 1988.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 159–173, 1989.

[AK90] S. Abiteboul and P. Kanellakis. Private communication, 1990.

[Apt88] Krzysztof R. Apt. Introduction to logic programming. Technical Report TR-87-35, (revised), Dept. of Computer Science, Univ. of Texas at Austin, July 1988. to appear in *Handbook of Theoretical Computer Science*.

[AV87] Serge Abiteboul and Victor Vianu. A transaction language complete for database update and specification. In *Proc. ACM Symp. on Principles of Database Systems*, pages 260–268, 1987.

[AV88a] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. Technical Report 900, INRIA, Sept. 1988.

[AV88b] Serge Abiteboul and Victor Vianu. Procedural and declarative database update languages. In *Proc. ACM Symp. on Principles of Database Systems*, 1988.

[Bee89] Catriel Beeri. Formal models for object oriented databases. In *Proc. of First Intl. Conf. on Deductive and Object-Oriented Databases*, 1989.

[BM81] P. Buneman and A. Motro. Constructing superviews. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 56–64, 1981.

[CFP84] M.A. Casanova, R. Fagin, and C.H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *J. Comput. Syst. Sci.*, 28(1):29–59, 1984.

[CH80] Ashok K. Chandra and David Harel. Computable queries for relational data bases. *J. Comput. Syst. Sci.*, 21(2):156–178, Oct. 1980.

[Che76] P.P. Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, March 1976.

[CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 77–90, 1977.

[Coh86] Don Cohen. Programming by specification and annotation. In *Proc. of AAAI*, 1986.

[Coh89] Don Cohen. Compiling complex database transition triggers. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 225–234, 1989.

[CW89] Weidong Chen and David S. Warren. C-Logic of complex objects. In *Proc. ACM Symp. on Principles of Database Systems*, pages 369–378, 1989.

[Day89] Umeshwar Dayal. Queries and views in an object-oriented data model. In *Proc. of Second Intl. Workshop on Database Programming Languages*, pages 80–102. Morgan Kaufmann, Los Altos, CA, June 1989.

[DH84] U. Dayal and H.Y. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. on Software Engineering*, SE-10(6):628–644, 1984.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HM85] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Trans. on Office Information Systems*, 3(3):253–278, July 1985.

[Hul86] R. Hull. Relative information capacity of simple relational schemata. *SIAM Journal of Computing*, 15(3):856–886, August 1986.

[Hul89] R. Hull. Four views of complex objects: A sophisticate's introduction. In S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 87–116. Springer-Verlag LNCS 361, 1989.

[HWW90] Richard Hull, Surjatini Widjojo, and Dave Wile. Specificational approach to database transformation. Technical report, USC/Information Sciences Institute, February 1990.

[HY91] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers, 1991. in preparation.

[Jac89] Dean Jacobs. A type system for algebraic database programming languages. In *Proc. of Second Intl. Workshop on Database Programming Languages*, June 1989.

[JK84] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.*, 28:167–189, 1984.

[KC86] S. Khoshafian and G. Copeland. Object identity. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 406–416, 1986.

[Klu80] A. Klug. Calculating constraints on relational expressions. *ACM Trans. on Database Systems*, 5(3):260–290, September 1980.

[KP82] Anthony Klug and Rod Price. Determining view dependencies using tableaux. *ACM Trans. on Database Systems*, 7(3):361–380, Sept. 1982.

[KV84] Gabriel M. Kuper and Moshe Y. Vardi. A new approach to database logic. In *Proc. ACM Symp. on Principles of Database Systems*, pages 86–96, 1984.

[KW89] M. Kifer and James Wu. A logic for object-oriented logic programming (Maier's o-logic revisited). In *Proc. ACM Symp. on Principles of Database Systems*, pages 379–393, 1989.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer-Verlag, Berlin, 1987.

[LV87] Peter Lyngbaek and Victor Vianu. Mapping a semantic database model to the relational model. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1987.

[Mai86] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Databases and Logic Prograamming*, pages 6–26, Washington, D.C., August 1986.

[Mit83] J. C. Mitchell. The implication problem for functional and inclusion dependencies. *Information and Control*, 56:154–175, 1983.

[Mot87] A. Motro. Superviews: Virtual integration of multiple databases. *IEEE Transactions on Software Engineering*, SE-13(7), July 1987.

[NT89] S. Naqvi and S. Tsur. *A Logical Langauge for Data and Knowledge Bases*. Computer Science Press, New York, 1989.

[RKS88] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, Dec. 1988.

[She90] Yeh-Heng Sheng. IDLOG: Extending the expressive power of deductive database languages. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 54–63, May 1990.

[Shi81] D. Shipman. The functional model and the data language DAPLEX. *ACM Trans. on Database Systems*, 6(1):140–173, 1981.

[Ull87] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems*. Computer Science Press, Potomac, Maryland, 1987.

[Var82] Moshe Y. Vardi. The complexity of relational query languages. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 137–146, 1982.

[vG86] A. van Gelder. Negation as failure using tight derivations for general logic programs. In *Proc. of Workshop on Foundations of Deductive Databases and Logic Programming*, 1986.

[WHW89] Surjatini Widjojo, Richard Hull, and Dave Wile. Distributed Information Sharing using WorldBase. *IEEE Office Knowledge Engineering*, 3(2):17–26, August 1989.