Evaluation Aspects of an Object-oriented Deductive Database Language

Georg Lausen and Beate Marx
Fakultät für Mathematik und Informatik
Universität Mannheim, W-6800 Mannheim, Germany
e-mail: {lausen, marx}@pi3.informatik.uni-mannheim.dbp.de

Abstract

Recently, F-logic [KL89, KLW92] has been proposed as an attempt to extend deductive databases by typical concepts of object-oriented languages. Among these concepts are complex objects, (term-based) object identity, methods, classes, typing, inheritance and browsing. In [KLW92] syntax and model-theoretic semantics is discussed; however many algorithmic aspects which arise when computing the corresponding models are left open. In this paper we start to bridge this gap. Several topics in the context of the evaluation of programs are discussed in detail; among these are weak recursion, global stratification and dynamic type-checking.

1 Introduction

Over the past few years object-oriented database systems have been receiving a lot of attention from both experimental and theoretical views. Since pure object oriented systems suffer from a lack of formal semantics which traditionally was considered to be important for database languages, there have been several attempts to combine object orientation and deductive databases (e.g. [AK89, KW89, KL89, KLW92, CW89, Mai86, HY90, AG87, BNST87]).

In this paper we continue the research direction started by Maiers O-logic [Mai86] and later extended by O-logic [KW89], C-logic [CW89] and F-logic [KL89, KLW92]. Among these approaches F-logic is the most elaborated one. In contrast to [AK89, HY90, AG87, BNST87] F-logic is a logic with function symbols. This allows a flexible syntax which naturally supports the creation of new objects, the definition of user-defined data-types, the parametrization of classes, etc. As a distinctive feature, F-logic has a higher-order syntax with first-order semantics; querying and browsing of the database is integrated into the same language.

In [KLW92] syntax and model-theoretic semantics is discussed; however many algorithmic aspects which arise when computing the corresponding models are left open. In this paper we start to bridge this gap. We show how the definition of class-hierarchies, typing and monotonic inheritance may be incorporated into a bottom-up evaluation strategy. As F-logic allows function-symbols, even for safe programs infinite relations may be derived, if the program is recursive. Therefore, in addition to safety, programs must be weakly recursive. In case recursion involves functional methods in a certain way, we can show that previously introduced definitions for weak recursion can be weakened. We introduce a unification-based test for global stratification and discuss various concepts to achieve type correctness.

The structure of the paper is as follows: Section 2 gives a short overview of syntax and semantics of F-logic, Section 3 deals with the evaluation of programs and Section 4 discusses aspects of type checking; Section 5 concludes the paper.

2 Syntax and Semantics

In the following two subsections we introduce syntax and semantics of F-logic [KLW92]. Here we can simplify the presentation, because we are only interested in rule-programs under Herbrand-semantics. Further, we restrict our discussion to monotonic inheritance: the non-monotonic case is discussed in [LU]. We assume some familiarity with first-order predicate logic and with Datalog^{neg} programs with function symbols (e.g., see [Llo87, Ull88]).

2.1 Syntax

The alphabet of an F-logic language $\mathcal{L}$ consists of (1) a set of *object constructors* $\mathcal{F}$, (2) a set $\wp$ of *predicate symbols*, (3) an infinite set of *variables* $\mathcal{V}$ and (4) usual logical connectives and quantifiers $\wedge, \vee, \forall, \exists, \neg, \longleftarrow$ etc. [KLW92]. Object constructors are function symbols. Each function symbol has an arity; symbols of arity 0 play the role of constant symbols, symbols of arity $\geq$ 1 are used to construct new objects from simpler ones. An *id-term* is a term composed of function symbols and variables in the usual way. The set of all ground id-terms is denoted by $\mathcal{F}^*$. Conceptually, ground id-terms should be perceived as *object-denotations*. While id-terms correspond to terms in first-order predicate logic, F-terms and P-terms (as introduced below) are the atomic formulae. In contrast to [KLW92], we do not introduce molecular terms. Every F-logic object (represented by an id-term) can be viewed as an entity or a relationship, a class or a method, depending on the syntactic position in a formula. In its role as a method, an object can either be *single valued* (also called *functional*) or *set-valued*. The way it has to be considered is determined by the context. In the sequel we will use names starting with lower-case letters to denote ground terms, and names starting with capital letters to denote terms that may be non-ground.

An *F-term* is one of the following:

- An *is-a* F-term, $P : Q$, where P and Q are id-terms; is-a terms are used to define class hierarchies and class extensions. P represents an object, resp. a (sub-)class, Q represents a (super-)class.

- A *data* F-term,
  $P[FunM \@ Q_1, \ldots, Q_k \rightarrow T]$ defining a functional method $FunM$, or $P[SetM \@ Q_1, \ldots, Q_k \twoheadrightarrow \{S_1, \ldots, S_m\}]$ defining a set-valued method $SetM$ on object P. T and the S_i represent the results returned by the respective methods *FunM* and *SetM* when invoked in the context of the object P on the arguments $Q_1, \ldots, Q_k$. P, $FunM$ and $SetM$, $Q_1, \ldots, Q_k$, $S_1, \ldots, S_m$ and T are id-terms.

- A *signature* F-term,
  $P[FunM \@ Q_1, \ldots, Q_k \Rightarrow \{A_1, \ldots, A_n\}]$, or $P[SetM \@ Q_1, \ldots, Q_k \Rightarrow\!\!\!> \{B_1, \ldots, B_m\}]$. The A_i and B_j represent classes which are the types of the results returned by the respective methods *FunM* and *SetM*, when

invoked in the context of an object of class P on arguments in classes $Q_1, \ldots, Q_k$. Here classes again are used as types. P, $FunM$ and $SetM$, $Q_1, \ldots, Q_k$, $A_1, \ldots, A_n$, and $B_1, \ldots, B_m$ are id-terms.

The following example illustrates these constructs. The first term defines a signature and states that *children* is a set-valued method that is applicable to objects belonging to class *person* resulting in a set of objects also belonging to class *person*. The second term respectively defines a signature for the method *age*. The third term states that *john* belongs to class *person* and the fourth and fifth term define actual values as results of methods *children* and *age* for the object *john*.

$$\begin{array}{l} person[children \Rightarrow\!\!\!> \{person\}] \\ person[age \Rightarrow \{integer\}] \\ john : person \\ john[children \rightarrow\!\!\!> \{sally, bob\}] \\ john[age \rightarrow 24] \end{array} \qquad (1)$$

Intuitively, a data F-term is a statement about an object asserting that it has properties specified by methods. A signature F-term specifies typing constraints on objects in the respective class. As types we allow standard classes, as *integer, string*, etc., and user-defined classes which may be defined by certain rules.

In addition to terms predicates are part of the language. If $p \in \wp$ is an n-ary predicate symbol and $T_1, \ldots, T_n$ are id-terms, then $p(T_1, \ldots, T_n)$ is a *predicate term* (abbrev. P-term). *Equality* of objects is expressed with the infix predicate $\doteq$, e.g., *john*$\doteq$*father(sally)*. The interpretation of the equality predicate is fixed in the usual way [CL73].

An *F-rule* is a clause

$$H \longleftarrow B_1 \wedge \ldots \wedge B_k \wedge B_{k+1} \wedge \ldots \wedge B_{k+l},$$

where H and the B_i, $i = 1, \ldots, k+l$ are F-terms or P-terms; H is called the *head* of the rule and $B_1 \ldots B_{k+l}$ the *body*. $B_1, \ldots, B_k$ are not negated, $B_{k+1} \ldots B_{k+l}$ are negated terms; if $l = 0$ a rule is called *Horn*. A rule with an empty body is called a *fact*. An *F-logic program* is a finite set of (implicitly) $\forall$-quantified F-rules; in our simplified setting the equality-predicate is not allowed to occur in the head of a rule.

2.2 Semantics

For an F-logic language $\mathcal{L}$, its semantic structure, I, is a tuple $\langle U, \preceq_U, I_\mathcal{F}, I_\wp, I_\rightarrow, I_{\twoheadrightarrow}, I_\Rightarrow, I_{\Rrightarrow} \rangle$. Here U is the domain of the interpretation and $\preceq_U$ is a partial order on U. $I_\mathcal{F}$ interprets k-ary object constructors, i.e., the elements of $\mathcal{F}$, as functions from U^k to U. Predicate symbols are interpreted by $I_\wp$ as relations on U of the appropriate arities. Methods are interpreted by $I_\rightarrow, I_{\twoheadrightarrow}, I_\Rightarrow, I_{\Rrightarrow}$ as mappings from U to partial functions from U^{l+1} to U, where l is the arity of the corresponding method. Semantic aspects of methods are expressed by special properties of these partial functions. For a detailed description see [KLW92]; for the topics of this paper Herbrand semantics is sufficient which is introduced below.

A *variable assignment*, ν, is a mapping from the set of variables, $\mathcal{V}$, to the domain of U. Let I be a semantic structure and ν a variable assignment. Intuitively, an F-term T is *true* under a semantic structure I with respect to variable assignment ν, denoted $I \models_\nu T$, if and only if I contains an object $\nu(T)$ with properties specified in $\nu(T)$. We introduce the notion of truth with respect to Herbrand interpretations.

2.2.1 Herbrand Interpretations

Given an F-Logic language $\mathcal{L}$ with $\mathcal{F}$ as its set of function symbols and $\wp$ as its set of predicate symbols, its *Herbrand universe* is $\mathcal{F}^*$ - the set of all ground id-terms. The *Herbrand Base*, $\mathcal{HB}$, is the set of all ground atoms of the F-terms, equality and P-terms. *Atoms* are defined as follows. Every is-a F-term and data F-term of a functional method is also an atom. A data F-term of a set-valued method is an atom if it is either of the form $P[SetM \,@\, Q_1, \ldots, Q_k \twoheadrightarrow \{S\}]$, or $P[SetM \,@\, Q_1, \ldots, Q_k \twoheadrightarrow \{\}]$. The atoms of a set-valued F-term are those which can be derived from it, i.e., their result is a subset of the result of the given F-term. Signature terms are treated analogously.

Herbrand interpretations are subsets of the Herbrand base; truth in Herbrand interpretations is defined as follows:

- A ground term (an F-term or a P-term) t is *true* in an Herbrand interpretation H, if for all atoms t' of t it holds $t' \in H$,

- A ground negative term $\neg t$ is *true* in H, if for at least one atom t' of t it holds $t' \notin H$,

- A rule $r = A \longleftarrow B_1 \wedge \ldots \wedge B_{k+l}$ is *true* in H, if for all ground instances $r\nu$ of r, ν a variable assignment, either $A\nu$ is true in H or at least one of the $B_i\nu$ is not true in H.

Now, let H be a subset of $\mathcal{HB}$. Then H is an *F-Herbrand interpretation* of $\mathcal{L}$ if it satisfies the equality axioms [CL73] w.r.t. $\doteq$ and in addition the following *closure properties* that reflect the properties of a semantic structure (cf. [KLW92]):

- ISA:

  ISA reflexivity: $p : p \in H$;
  ISA transitivity: If $p : q, q : r \in H$, then $p : r \in H$;
  ISA acyclicity: If $p : q, q : p \in H$, then $p \doteq q \in H$.

- Signature:

  Type inheritance:
  If $p[funM \circledcirc q_1, \ldots, q_k \Rightarrow \{s\}], r : p \in H$,then
  $r[funM \circledcirc q_1, \ldots, q_k \Rightarrow \{s\}] \in H$;
  If $p[setM \circledcirc q_1, \ldots, q_k \Rightarrow\!\!\!> \{s\}], r : p \in H$,then
  $r[setM \circledcirc q_1, \ldots, q_k \Rightarrow\!\!\!> \{s\}] \in H$.
  Argument subtyping:
  If $p[funM \circledcirc q_1, \ldots, q_i, \ldots, q_k \Rightarrow \{s\}], q_i' : q_i \in H$ then
  $p[funM \circledcirc q_1, \ldots, q_i', \ldots, q_k \Rightarrow \{s\}] \in H$;
  If $p[setM \circledcirc q_1, \ldots, q_i, \ldots, q_k \Rightarrow\!\!\!> \{s\}], q_i' : q_i \in H$ then
  $p[setM \circledcirc q_1, \ldots, q_i', \ldots, q_k \Rightarrow\!\!\!> \{s\}] \in H$.
  Range supertyping:
  If $p[funM \circledcirc q_1, \ldots, q_k \Rightarrow \{r\}], r : s \in H$ then
  $p[funM \circledcirc q_1, \ldots, q_k \Rightarrow \{s\}] \in H$;
  If $p[setM \circledcirc q_1, \ldots, q_k \Rightarrow\!\!\!> \{r\}], r : s \in H$ then
  $p[setM \circledcirc q_1, \ldots, q_k \Rightarrow\!\!\!> \{s\}] \in H$.

- Data F-terms:

  functionality:
  If $p[funM \circledcirc q_1, \ldots, q_k \rightarrow r_1], p[funM \circledcirc q_1, \ldots, q_k \rightarrow r_2] \in H$, then
  $r_1 \doteq r_2 \in H$.

- Id-terms:

  For every id-term t, $t[\,] \in H$.

The notion of type correctness in F-logic is as follows. A program P is *well-typed* with respect to a Herbrand interpretation H, if, in addition, the following properties hold:

Signature:

Well-typing:

If $p[funM \@ q_1, \ldots, q_k \rightarrow q],\ p[funM \@ q_1, \ldots, q_k \Rightarrow \{r\}] \in H$ then $q : r \in H$;

if $p[setM \@ q_1, \ldots, q_k \twoheadrightarrow \{q\}],\ p[setM \@ q_1, \ldots, q_k \Rightarrow\!\!\!> \{r\}] \in H$ then $q : r \in H$;

if $p[funM \@ q_1, \ldots, q_k \rightarrow q] \in H$ then $p[funM \@ q_1, \ldots, q_k \Rightarrow \{\}] \in H$;

if $p[setM \@ q_1, \ldots, q_k \twoheadrightarrow \{q\}] \in H$ then $p[setM \@ q_1, \ldots, q_k \Rightarrow\!\!\!> \{\}] \in H$.

Truth in F-Herbrand interpretations is analogous to the Herbrand case.

Given an F-logic program P, a Herbrand interpretation M is a *Herbrand model* for P, if all rules of P are true in M. M is an *F-Herbrand model* if, in addition, the closure properties hold. M is a minimal Herbrand model (resp. minimal F-Herbrand model) if, for all other models (F-models) N, whenever $N \subseteq M$ then $N = M$. If all rules in a program P are Horn, a unique minimal model exists. In the sequel we are interested in minimal F-models for general programs.

2.2.2 Closure and inheritance rules

For every F-logic program P the closure properties listed above may be restated as a finite set of Horn rules, which we call *closure rules*; they are denoted by the symbol $\overline{P}$. Thus, if P has a minimal model, $P \cup \overline{P}$ has a minimal model. Or in other terms: M is a minimal F-model of P if and only if M is a minimal model of $P \cup \overline{P}$. For example, the ISA properties are stated by the rules

$$\begin{aligned} P : P &\leftarrow \\ P \doteq Q &\leftarrow P : Q \ \wedge\ Q : P \\ P : Q &\leftarrow P : R \ \wedge\ R : Q. \end{aligned} \tag{2}$$

Giving another example, type inheritance for functional methods may be stated as

$$P[M \@ X_1, \ldots, X_k \Rightarrow \{Y\}] \leftarrow P : Q \ \wedge\ Q[M \@ X_1, \ldots, X_k \Rightarrow \{Y\}] \tag{3}$$

For every arity of a functional method occurring in P, such a rule is needed. The remaining closure properties are transformed to Horn rules in a similar way.

In the current paper we only consider monotonic inheritance. Monotonic inheritance may be stated as a set of Horn rules in a similar way as inheritance of signatures, i.e., for every arity of a functional (resp. set-valued) method a rule

$$P[M @ X_1, \ldots, X_k \rightarrow Y] \longleftarrow P : Q \ \wedge \ Q[M @ X_1, \ldots, X_k \rightarrow Y] \qquad (4)$$

is needed. Throughout the following sections P^{inh} will denote the set of *inheritance rules* belonging to program P .

3 Evaluation of programs

3.1 Extending the T-operator

We denote T_P the *immediate consequence operator* (which is defined as in [ABW89]). Intuitively, $T_P(I)$ is the set of immediate conclusions from an interpretation I, i.e., those which can be obtained by applying each rule from P exactly once. For Horn programs P, $T_P \uparrow \omega(\emptyset)$ (the least fixpoint of the operator T_P starting with the empty interpretation), and the minimal model of P, M_P, coincide.

For a Horn F-logic program P the minimal F-model, M_P, equals $T_{P \cup \overline{P} \cup P^{inh}} \uparrow \omega(\emptyset)$. Since the set of closure (resp. inheritance) rules is essentially the same for all F-logic programs (it only differs with respect to the arity of methods), we treat program and closure (resp. inheritance) rules separately and define the *F-completion* operator $\overline{T}$ as an abstraction of $T_{\overline{P} \cup P^{inh}}$; given any interpretation I for a program P, the operator $\overline{T}$ completes I to an F-interpretation of P. The minimal F-model of a Horn program P is thus computed as $(\overline{T}(T_P)) \uparrow \omega(\emptyset)$.

Note, that, in the general case, an F-interpretation of a program P may not satisfy the well-typing properties (see Section 2.2.1). The problem of type-checking will be discussed in Section 4.

After each application of the T_P-operator a functionality check should be performed. We believe, that equalities, that are introduced on account of the

functionality or the ISA-antisymmetry property, are usually not intended by the user (cf. [KLW92]). In fact, if during program evaluation (after applying T_P and before applying the succeeding $\overline{T}$), an interpretation contains facts, say

$$obj[functional\_method \rightarrow a],\ obj[functional\_method \rightarrow b] \tag{5}$$

but not

$$a \doteq b, \tag{6}$$

the user has probably specified an inconsistent program and may find it helpful, if the system responds with a warning instead of producing an unintended equality.

To detect these inconsistencies during program evaluation we introduce a slightly modified completion operator: The derivation of equalities on account of the functionality and the ISA-antisymmetry property is blocked; we call the modified operator $\overline{T}^{\text{block}}$. The complementary operator, i.e., the operator that completes an interpretation with respect to functionality and ISA-antisymmetry, is called $\overline{T}^{\text{unblock}}$. In terms of the operators $\overline{T}^{\text{block}}$ and $\overline{T}^{\text{unblock}}$ the minimal F-model of a program P may be computed by evaluating $(\overline{T}^{\text{unblock}}(\overline{T}^{\text{block}}(T_P))) \uparrow \omega(\emptyset)$. If at any iteration step the operator $\overline{T}^{\text{unblock}}$ adds new terms to the interpretation computed thus far the system will notify these inconsistencies to the user.

3.2 Safety and weak recursive programs

Informally, a logic program is called *safe*, if it is guaranteed that no rule creates an infinite relation from finite ones. Usually this is achieved by placing some syntactic restrictions on the rules. For Datalog programs the concept of *limited* variables is introduced (e.g., [Ull88]). A rule is defined to be safe, if every variable occurring somewhere in this rule is limited. The notion of safety can be adopted to F-logic programs in an obvious way. Henceforth we only consider safe programs. However, even safe F-logic programs may have to be rejected because their bottom-up evaluation may not terminate due to an infinite creation of id-terms. The latter takes place, if there is recursion over rules containing object constructors (i.e. function symbols) in the head term. This problem is known from Datalog-like languages with function symbols (see e.g. [HY90]), yet, on account of the syntax of F-logic (variables

may occur at any position of an F-term) and because of the functionality of methods new aspects arise. Functional methods, when invoked on an object, have unique results. Thus infinite object-id creation may not be propagated over result positions of functional methods. This fact allows to weaken the safety restrictions concerning the use of object constructors with respect to other Datalog-like languages with function symbols, i.e., F-logic programs may be accepted in cases where corresponding Datalog programs would be rejected.

In analogy to Hull and Yoshikawa [HY90] F-logic programs are called *weakly recursive* if there is no recursion through object constructors. The following examples show F-logic rules and programs respectively, that are not weakly recursive:

$$child(X)[address \rightarrow A] \longleftarrow X[address \rightarrow A] \tag{7}$$

$$X[related \twoheadrightarrow \{child(Y)\}] \longleftarrow X : person \wedge\ X[related \twoheadrightarrow \{Y\}] \tag{8}$$

$$\begin{aligned} f(X) : some\_class &\longleftarrow Y : some\_other\_class \wedge\ Y[method \twoheadrightarrow \{X\}] \\ Y[method \twoheadrightarrow \{X\}] &\longleftarrow X : some\_class \wedge Y : some\_other\_class \end{aligned} \tag{9}$$

$$class[method \twoheadrightarrow \{f(Z)\}] \longleftarrow object : class \wedge object[method \twoheadrightarrow \{Z\}] \tag{10}$$

Note, that the last rule, (10), shows object creation interleaving with monotonic inheritance of methods: it is an example for implicit recursion through object generation.

To detect recursion through object creation occurring within an F-logic program P we introduce the *safety graph* $\mathcal{S}(P)$ as follows: The nodes of the graph are the id-term positions[1] in terms occurring in $P \cup \overline{P} \cup P^{inh}$; the arcs of $\mathcal{S}(P)$ reflect the propagation of newly created objects during program evaluation. The closure rules and the inheritance rules, $\overline{P} \cup P^{inh}$, must be integrated into $\mathcal{S}(P)$ since recursion may be implicit as in Rule (10).

In the following we outline an informal specification of the procedure which, given a program P, constructs $\mathcal{S}(P)$. During an initializing step, only rules having a non-ground id-term t with at least one function symbol in the head (i.e., rules where an object creation takes place) must be considered.

[1] An id-term position is any syntactical position within an F-term or a P-term, where an id-term (or a variable) may occur.

Considering just these rules during the first step ensures that all cycles, that may finally occur, are indeed cycles over object creation. Let $X_1, \ldots, X_k$ be the variables that occur in such t. To reflect the flow of the X_i into t, for each t'_j occurring somewhere in the body of that rule that contains at least one of the X_i an arc is introduced from t'_j to t. Consider Example (7): here an arc is drawn from the X at the object position in the F-term of the body to *child(X)* in the head.

For the remaining part of the procedure all rules of $P \cup \overline{P} \cup P^{inh}$ must be considered. Since we are interested in the propagation of object-ids (i.e., id-terms) that are built using function symbols we have to iteratively check nodes with ingoing arcs. Two cases must be distinguished:

(S1) The node is an id-term t in the head h of some rule.
For all terms b_i in the body of the same or any other rule that unify with h, draw an arc from t to the id-term at the corresponding position in b_i.
In Example (7) the head unifies with the body, thus an arc is drawn from *child(X)* to X in $X[address \rightarrow A]$.

(S2) The node is an id-term t in a term b in the body of some rule.
Let $X_1, \ldots, X_k$ be the variables of t, which also occur in the head h of the same rule. Let $t'_1, \ldots, t'_l$ be the id-terms containing at least one of the X_i's. Draw arcs from t to each of the t'_j, respectively.
In case the id-term t is part of an equality term, $t \doteq t'$, draw an additional arc from t to t'.

The procedure stops, when no more arcs can be entered into the graph (because of (S1) or (S2)).

In Figure 1 the safety graph of Example (9) is shown. The ellipses group together nodes which belong to the same term, they are labeled with the F-term they correspond to. Nodes that are not connected to any other nodes do not contribute to the creation of new objects.

If, for a program P, $\mathcal{S}(P)$ does not contain any cycle, the program is free of recursion through object generation and thus weakly recursive. In the rest of this section we will show, that the functionality property of F-logic programs allows a weaker condition for weak recursion.

Consider the following example:

$$p(X, Y) \longleftarrow X : person \wedge \ X[best\_friend \rightarrow Y]$$

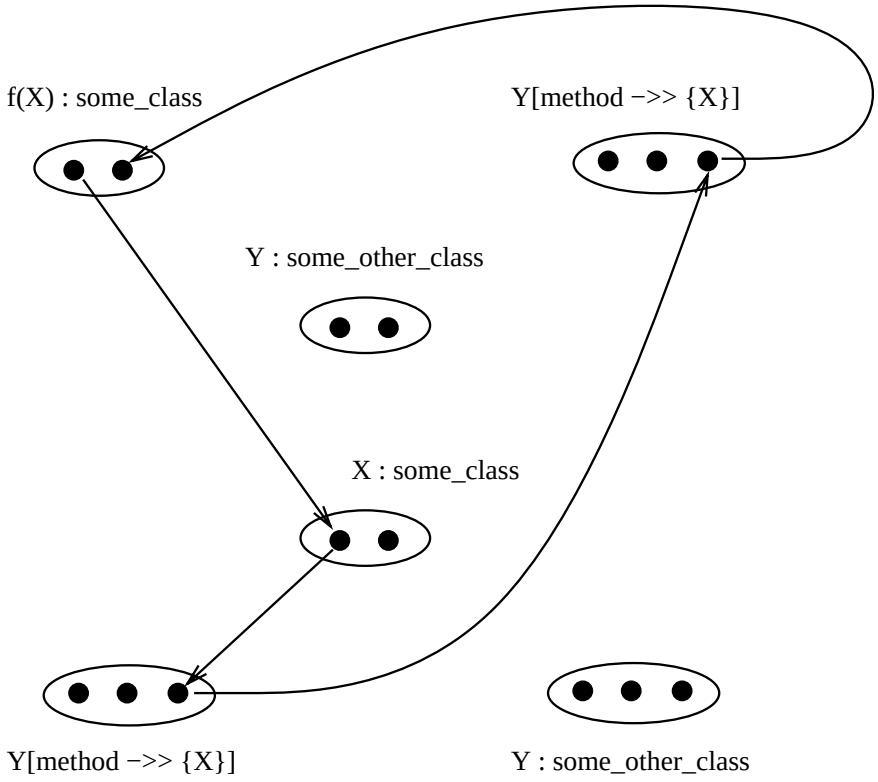

Figure 1: Safety graph

$$X[best\_friend \rightarrow child(Y)] \longleftarrow p(X,Y) \wedge q(\ldots) \tag{11}$$

Here the method *best_friend* is functional, and thus the result of applying the method to any object of class *person* must be unique. The evaluation of these rules may or may not result in an infinite number of F-terms of the form $X[best\_friend \rightarrow child(child(...(Y)...)]$, depending on the predicate q. In our framework such a program will not be rejected. Since during program evaluation the functionality of methods must be checked anyway (see Section 3.1), the admission of this rule, independent of predicate q, will not lead to an infinite computation. Therefore, we can expect that a larger class of programs can be evaluated, because the syntactic safety check is replaced by a semantic one.

Consider one more example:

$$X[related \twoheadrightarrow \{child(Y)\}] \longleftarrow X : person \wedge X[related \twoheadrightarrow \{Y\}] \wedge Y : person \tag{12}$$

Rule (12) is similar to Rule (8) belonging to the examples for rules, which are not weakly recursive. The former Rule (8) obviously is unsafe, since for every variable assignment for Y, a new id-term $child(Y)$ is created, which may be assigned to Y during the next evaluation step, and so on. Rule (12) differs from Rule (8): the variable Y also occurs in another positive term in the body of the rule. Here after one application of the rule, the newly created id-term $child(\ldots)$ may only then be assigned to Y during the next step, if $child(\ldots) : person$ holds. Provided the extension of class *person* is finite, or in other terms, *person* is a finite type, the evaluation of Rule (12) terminates.

To capture this intuition and the restrictions on cause of functionality, we extend the definition of the safety graph. Let $\mathcal{S}(P)$ be a safety graph according to (S1) and (S2). We now derive the *reduced* safety graph $\mathcal{S}^*(P)$ by applying the following two steps on $\mathcal{S}(P)$:

(S3) Let $X_1, \ldots, X_k$ be the variables of an id-term t, where t occurs in the body of some rule r. If each X_i is of a finite type, then any arc starting from t is removed.

(S4) Let t be an id-term which occurs in the result position of a functional method of an F-term f. Let $t \rightarrow t'$ be an arc in $\mathcal{S}(P)$ and let $t \rightarrow t' \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_k \rightarrow t$, $k \geq 0$, be any cycle, where the t_i are id-terms ocurring in F-terms f_i, $i = 1, \ldots, k$. If for every f_i which unifies with $f\ t_i$ occurs at the result position, then $t \rightarrow t'$ is removed.

Condition (S4) may seem unnecessarily restrictive. Yet, it is not sufficient to ignore cycles containing at least one node that corresponds to the result position of a functional method. As the following example shows, we must assure that the respective object (resp. the method or its arguments) does not change in such a way, that whenever a new result is created it is assigned a new object (resp. method or argument) in such a way, that there is a cyclic shift of object-ids:

$$f(Y)[method \rightarrow X] \longleftarrow X[method \rightarrow Y]. \tag{13}$$

Starting with a fact $a[method \rightarrow b]$ the following facts will be computed successively: $f(b)[method \rightarrow a]$, $f(a)[method \rightarrow f(b)]$, $f(f(b))[method \rightarrow f(a)]$, $f(f(a))[method \rightarrow f(f(b))]$ and so on. Although the cycle corresponding to Rule (13) contains a node which corresponds to the result position of a functional method, it is not weakly recursive; this special case is captured by extension (S4).

We now are in the position to redefine weak recursion; a program P is called *weak recursive*, if its extended safety graph $\mathcal{S}^*(P)$ does not contain a cycle.

3.3 Perfect models of F-logic

Perfect model semantics for general logic programs [Prz89] can be extended to F-logic programs. Consider the following examples:

$$X[wants \twoheadrightarrow \{Y\}] \Longleftarrow \neg X[has \twoheadrightarrow \{Y\}] \tag{14}$$

and

$$peter[wants \twoheadrightarrow \{Y\}] \Longleftarrow \neg john[wants \twoheadrightarrow \{Y\}] \tag{15}$$

Note, that both rules are recursive; in (14) we have recursion over the object-position - in (15) over the method-position of the respective F-terms. Moreover, in both cases recursion involves a negated F-term. However, both cases are compatible with stratification, because we either know, that the methods involved are different, or that the respective objects are different. Thus, stratification for F-logic programs should be ensured with respect to complete F-terms (resp. P-terms), not just with respect to objects or methods as the analogy to Datalog might suggest. The following example shows that techniques based on predicate names (respectively, their analogue) to derive a (global) stratification cannot be applied directly, because there might be F-terms, which have nonground id-terms in all positions:

$$\begin{aligned} &married(sally, john)[shared\_methods \twoheadrightarrow \{address\}] \\ &X[M \rightarrow V] \Longleftarrow married(X,Y)[shared\_methods \twoheadrightarrow \{M\}] \wedge Y[M \rightarrow V] \\ &X[owns \twoheadrightarrow \{car\}] \Longleftarrow X[works \rightarrow T] \wedge \neg X[address \rightarrow T] \end{aligned} \tag{16}$$

To check whether an F-logic program P is globally stratified and to compute a global stratification (provided it exists), we introduce the *dependency graph* $\mathcal{D}(P)$ as follows: The nodes of $\mathcal{D}(P)$ are the terms of $P \cup \overline{P} \cup P^{inh}$.[2]

For every rule $r = A \longleftarrow B_1, \ldots B_n$ contained in $P \cup \overline{P} \cup P^{inh}$, we have arcs as follows:

(D1) From every term B_i occurring in the body of r, draw an arc to the head term A. If B_i occurs negated then the respective arc must be marked as negated.

(D2) For every term B in the body of a rule that unifies with A, draw an arc from A to B.

Note that in comparison to Datalog, we have to consider *unification arcs* (the second kind of arcs introduced above), since in F-logic variables may occur at any position of a term. Note further, that on account of closure and inheritance rules a lot of unification arcs are introduced to the graph. This may cause problems for the computation of the stratification of a program; we will consider this topic at the end of the section.

An F-logic program P is *globally stratified*, if $\mathcal{D}(P)$ has no cycle containing marked arcs.

Based on the concept of global stratification we now define an evaluation strategy for general programs as follows (see also [ABW89]):

Decompose the dependency graph $\mathcal{D}(P)$ into strongly connected components of maximum cardinality. Sort these components topologically; ambiguities may be solved in any way. Each component corresponds to a stratum of P (the assignment is done on account of the membership of a rule's head atom to a component), thus the ordering of the components induces an evaluation ordering on the rules of P. Evaluating the strata in this order yields the perfect model of P.

Consider once more Example (16). Figure 2 shows the relevant part of the corresponding dependency graph. The graph contains one negated arc that does not lie within a cycle, thus the program is globally stratified. The only cycle is enforced by the second rule; strata may thus be defined as follows:
$S_0 = \{married(sally, john)[shared\_methods \twoheadrightarrow \{address\}]\}$,

[2]For terms, that are textually equivalent modulo variable renamings, e.g., $X[M \rightarrow V]$ and $Y[M \rightarrow W]$, only one node may be introduced.

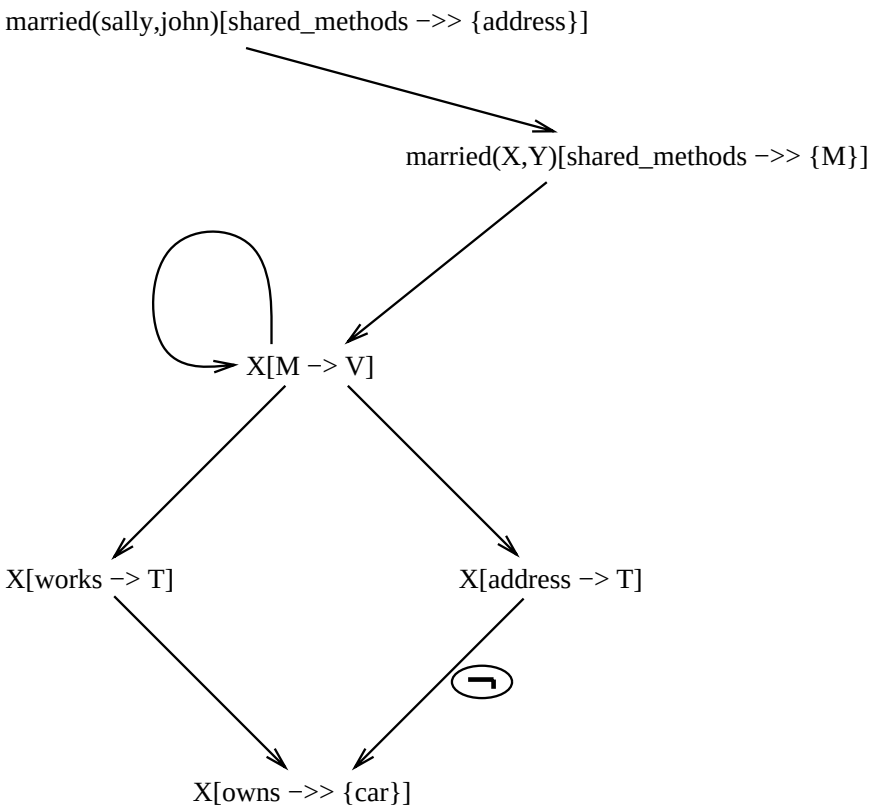

Figure 2: Dependency graph

$S_1 = \{X[M \rightarrow V] \longleftarrow married(X,Y)[shared\_methods \twoheadrightarrow \{M\}] \wedge Y[M \rightarrow V]\}$ and
$S_2 = \{X[owns \twoheadrightarrow \{car\}] \longleftarrow X[works \rightarrow T \wedge \neg X[address \rightarrow T]\}$.

The evaluation of the perfect model of a general program P is then performed by computing the fixpoint (with respect to the combined operator $\overline{T}(T)$) of each stratum, beginning with stratum S_0 and the empty set as input and carrying on in such a way that the fixpoint of stratum S_i is the input for the computation of the fixpoint of stratum S_{i+1} (see [Ull88, ABW89]).

3.4 Reducing the dependency graph

As mentioned before, including the closure and inheritance rules into the dependency graph leads to problems; the more arcs are introduced to the graph on behalf of unification (D2) the greater is the probability that cycles (and also negative cycles) may occur. The following example will show that unification arcs, introduced on behalf of closure (resp. inheritance) rules, very easily may cause unjustified cycles. Consider the following rule for monotonic inheritance:

$$P[M \rightarrow Y] \longleftarrow P : Q \wedge \ Q[M \rightarrow Y] \tag{17}$$

Any rule having terms with 0-ary functional methods in its body will cause an arc from the head of the inheritance rule to the respective term; any rule with head being of the respective form will cause an arc from its head to the respective term in the body of the closure rule. The impact on stratification may be demonstrated by the following program P.

$$\begin{array}{l} john : empl \\ john[lives \rightarrow munich] \\ john[works \rightarrow mannheim] \\ X[working\_address \rightarrow T] \longleftarrow X : empl \ \wedge \ X[works \rightarrow T] \\ X[home\_address \rightarrow T] \longleftarrow X : empl \ \wedge \ X[lives \rightarrow T] \ \wedge \ \neg X[works \rightarrow T] \end{array} \tag{18}$$

Figure 3 shows the dependency graph of Program (18); for the sake of simplicity we have omitted all closure and inheritance rules but Rule (17). According to the dependency graph P is not stratified since there is a cycle over a marked edge. P is rejected, although the intended meaning is quite clear and may be described as follows: *john* is an *employee*; thus he has 'Mannheim' as working_address and 'Munich' as home_address. A closer inspection of the dependency graph of Figure 3 shows that propagating the unifier computed at the unification arc ending in the inheritance rule along the cycle renders the unification arc starting at the inheritance rule invalid (since *home_address* does not unify with *works*). The cycle is thus not *stratification-relevant* and may be ignored. This example outlines a strategy that accepts a greater number of programs as being stratified than simply checking the dependency graph for cycles over marked edges and dismissing programs as suggested in the previous subsection: For each cycle containing

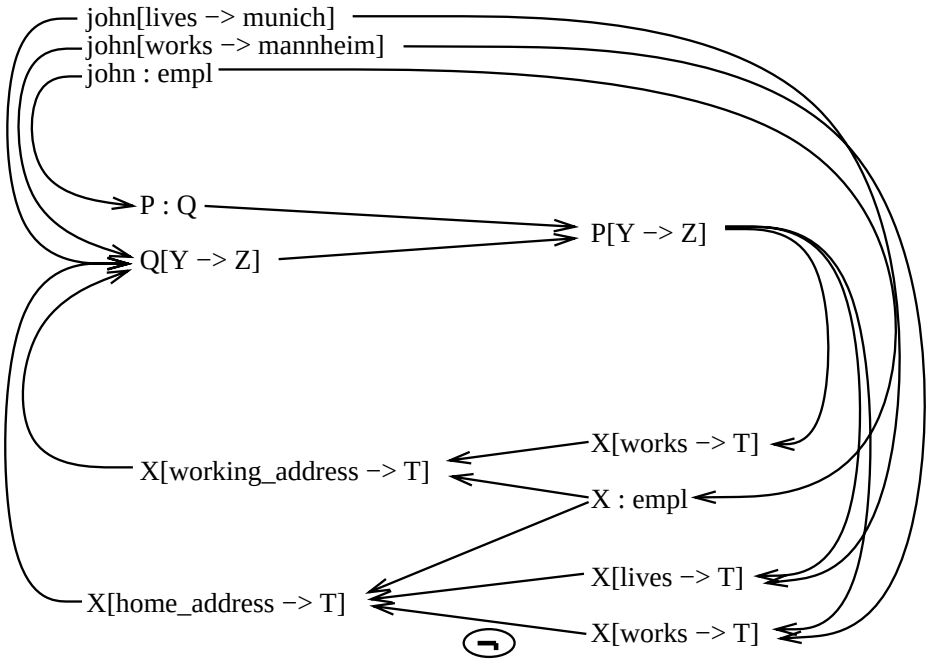

Figure 3: Dependency graph of Example 19

a marked edge (only these cycles are relevant for stratification) check whether propagating unifiers along the cycle invalidates unification arcs. If one of the negative cycles is stratification-relevant (for a formal definition see below) reject the program; otherwise, if none of the cycles is stratification-relevant, break the cycles (e.g., by ignoring the body-head arc belonging to the closure (resp. inheritance) rule), compute the stratification and, depending on the stratification, the perfect model as outlined in the previous subsection.

Consider one final example sketched in Figure 4, where a cycle involving the inheritance rule

$$P[M \rightarrow X] \longleftarrow P : Q \ \wedge \ Q[M \rightarrow X] \tag{19}$$

is indicated. Here propagation of unifiers fails, since *person* does not unify with *empl*. Nevertheless, because of inheritance of methods, the *empl*-term depends on the *person*-term, i.e., the cycle is relevant for stratification.

To handle these cases correctly we define *isa-unification* of id-terms as follows: An id-term t isa$\uparrow$-unifies with an id-term t' if either t unifies with t'

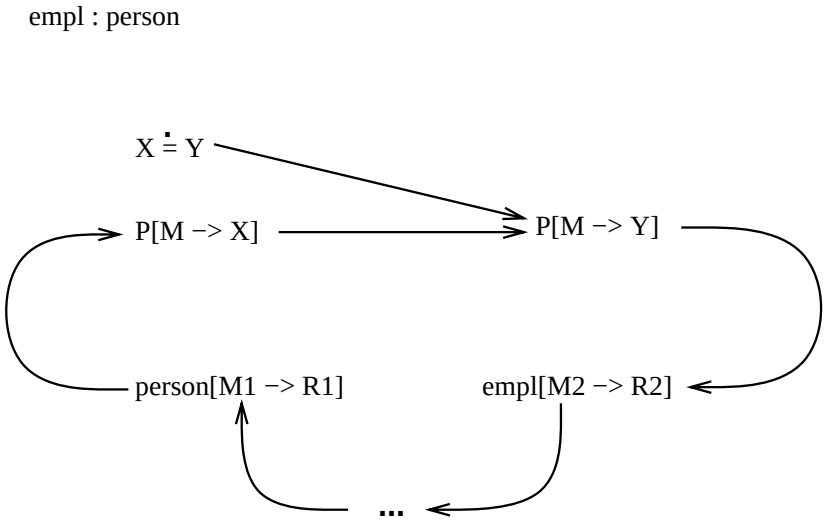

Figure 4: An example for isa-unification

or if for some $t^\star$ $t : t^\star$ holds and $t^\star$ unifies with t'.[3] Accordingly, t isa$\downarrow$-unifies with an id-term t' if either t unifies with t' or if for some $t^\star$ $t^\star : t$ holds and $t^\star$ unifies with t'. For two data F-terms $t = P[M \,@\, A_1, \ldots, A_k \rightarrow Q]$ and $t' = P'[M' \,@\, A'_1, \ldots, A'_k \rightarrow Q']$ t *isa-unifies* with t' if P isa$\uparrow$-unifies with P', M unifies with M', the A_i unify with the A'_i and Q unifies with Q'. For two signature F-terms $t = P[M \,@\, A_1, \ldots, A_k \Rightarrow \{Q\}]$ and $t' = P'[M' \,@\, A'_i, \ldots, A'_k \Rightarrow \{Q'\}]$ t *isa-unifies* with t' if P isa$\uparrow$-unifies with P', M unifies with M', the A_i's isa$\uparrow$-unify with the A'_i's and Q isa$\downarrow$-unifies with Q' (note that isa-unification of F-terms is asymmetric); this definition is extended to terms over set-valued methods in the obvious way. Any cycle in the dependency graph is *stratification-relevant* if for all closure (resp. inheritance) rules, whose corresponding arcs occur in the cycle, any F-term, which is a direct successor, isa-unifies with the respective direct predecessor F-terms. For the sake of efficiency we do not propagate unifiers along the whole cycle but consider just the neighbourhood of closure rules.

We can now weaken the definition of global stratification: An F-logic program is *globally stratified* if it does not contain a stratification-relevant

[3]Here we assume, that before a stratification is computed the complete isa-relationships are known. In Section4 we discuss corresponding evaluation frameworks.

cycle over a negative (marked) edge.

4 Type checking

Different to the treatment of types as it is done usually in object-oriented database systems, e.g., in [ACO90], F-logic allows reasoning about data and meta-data (i.e., types) within an integrated framework. Thus type checking may in general not be done at compile time, i.e., static type checking in the strong sense may not be applied to F-logic programs.

In this section we will classify F-logic programs with respect to their type checking properties. We therefore characterize the rules of a program P as follows: a rule is a *signature rule*, if its head is a signature F-term, it is an *ISA rule*, if its head is an ISA F-term and it is a *data rule* otherwise. We further call a rule *data dependent*, if either any data F-term or P-term occurs in the body of the rule, or (recursively) if any term in the body of the rule unifies with the head of a data dependent rule. Given a Program P, the set of all signature (resp. ISA) rules belonging to P is called P^{sig} (resp. P^{isa}).

We distinguish three classes of programs:

- A program P is *partially statically typed*, if $P^{\text{sig}} \cup P^{\text{isa}}$ is *data independent*. Note that signature rules may depend on ISA-terms and vice versa.

- A program P is *dynamically typed*, if P^{isa} is data independent and if in addition P is *signature stratified* (as defined below).

- All other programs belong to the class of programs where type-checking may only be applied after evaluating the program.

Consider the class of partially statically typed programs. A partially statically typed program P may be split into P^{type}, the union of P^{sig} and P^{isa}, and P^{data}. After computing the fixpoint for P^{type} all type information concerning P is known and static type checking may be invoked on P^{data}. We do not further elaborate on this topic, since in the current paper we are mainly interested in dynamically typed programs.

Next consider programs belonging to the class of *dynamically typed* programs. Intuitively, a program P is dynamically typed, if, whenever a data

term is computed during program evaluation[4], it may be immediately type-checked; particularly, information once established about type correctness of a data term should not be invalidated later. Since signature information is stated explicitly through signature rules and implicitly through inheritance (see Section 2.2), for dynamic type-checking it is necessary to know the complete hierarchy before computing the first data term, thus ISA rules must be data independent and they must be evaluated prior to all data rules. To check whether all relevant type information is available whenever a data-term is computed, we extend the dependency graph $\mathcal{D}(P)$ introduced in Section 3.3 to $\mathcal{D}^{\text{sig}}(P)$ by introducing *signature arcs* as follows:

(D3) For every pair consisting of a data- and a signature term both occurring either as a fact or in the head of a rule, check whether the latter may be *relevant* to the former. If this is the case insert a signature arc from the node corresponding to the signature term to the node corresponding to the data term.

A signature term having the form $obj[method@\overrightarrow{args}\ldots]$ is *relevant* to a data term having the form $obj'[method'@\overrightarrow{args'}\ldots]$, if there holds:

> obj unifies with an id-term t, obj' unifies with an id-term t, where $t' : t$, and $method$ and $method'$ may denote the same method, i.e., $method$ and $method'$ unify, the number of arguments is equal and both methods are either functional or set-valued.

A program P is *signature stratified*, if the extended dependency graph $\mathcal{D}^{\text{sig}}(P)$ has no cycles containing signature arcs. As stated before, signature stratified programs with data independent ISA-rules are dynamically typed. The evaluation strategy for general F-logic programs introduced in Section 3.3 may now be extended to integrate dynamic type checking by computing the strata of a program P with respect to $\mathcal{D}^{\text{sig}}(P)$. Then, while evaluating the rules in an order imposed by the ordering of the strata, each data term may be immediately type-checked; type checking may be integrated into the T-operator. Thus evaluating a program and type-checking of its results may be done within the same framework.

[4]We assume that programs are evaluated bottom-up.

5 Conclusion

In this paper we discuss several aspects concerning the evaluation of F-logic programs when techniques are applied which are known from deductive databases. We discuss weak recursive programs and we are able to show, that the knowledge about functionality of methods can be used to weaken previous definitions. The stratification of general programs in F-logic is complicated by the fact that variables may occur anywhere in an F-term. Stratification for Datalog programs is defined with respect to predicate symbols. In general, F-terms do not embody something (syntactically) equivalent to predicate symbols; we thus define stratification with respect to the complete terms in the rules. To this end we have to base the definition of a dependency graph on unification arcs (which are not needed in Datalog).

Each program is extended by a set of closure and inheritance rules. Yet, building a dependency graph over an extended program may increase the number of negative cycles (because of the great number of unification arcs that are introduced on account of the closure rules). We distinguish between relevant and irrelevant cycles on behalf of the propagation of unifiers over closure rules. A stratification is then computed with respect to relevant cycles.

Finally, we discuss dynamic typing and type checking. We introduce various concepts of type correctness. Since we are interested in dynamic typing we define certain criteria which must be fulfilled such that dynamic type checking can be applied. We finally show how to integrate dynamic type checking into a bottom-up evaluation strategy.

Acknowledgements: We would like to thank Jürgen Frohn, Michael Kramer and Heinz Uphoff for helpful comments and discussions on the topics of this paper.

References

[ABW89] Krzysztof R. Apt, Howard Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Founda-*

tions of Deductive Databases and Logic Programming, pages 1 – 77. Morgan Kaufmann, 1989.

[ACO90] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, chapter 2, pages 147 – 162. Morgan Kaufmann, 1990.

[AG87] Serge Abiteboul and Stephane Grumbach. COL: A logic-based language for complex objects. In *1st Workshop on Database Programming Languages*, pages 253 – 276, 1987.

[AK89] Serge Abiteboul and Paris Kanellakis. Object identity as a query language primitive. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 159 – 173, 1989.

[BNST87] Catriel Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL). Technical report, MCC, 1987.

[CL73] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[CW89] Weidong Chen and David S. Warren. C-logic of complex objects. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 369 – 378, 1989.

[HY90] Richard Hull and Masatoshi Yoshikawa. Ilog: Declarative creation and manipulation of object identifiers. In *Proceedings of the Intl. Conference on Very Large Data Bases*, pages 455 – 468, 1990.

[KL89] Michael Kifer and Georg Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 134 – 146, 1989.

[KLW92] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object oriented and frame-based languages. accepted for publication, 1992.

[KW89] Michael Kifer and James Wu. A logic for object-oriented programming (Maier's O-logic revisited). In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379 – 393, 1989.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd edition, 1987.

[LU] Georg Lausen and Heinz Uphoff. Aspects of inheritance in a rule language. Manuscript.

[Mai86] David Maier. A logic for objects. In *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6 – 26, 1986.

[Prz89] Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 191 – 216. Morgan Kaufmann, 1989.

[Ull88] J.D. Ullman. *Principles of Database and Knowledgebase Systems.* Computer Science Press, 1988.