Semi-naive Evaluation of F-logic Programs
Technical Report 85

Christian Schlepphorst

Institut für Informatik, Albert-Ludwigs-Universität
79110 Freiburg, Germany
schlepph@informatik.uni-freiburg.de

Abstract

The semi-naive evaluation strategy is one of the most common optimization techniques for enhancing bottom-up evaluation in the field of deductive databases. It is almost a must for a deductive database system to have this strategy implemented. F-logic is a powerful language combining the advantages of deductive databases with the rich modelling capabilities of object oriented concepts (objects, methods, class hierarchy, non-monotonic inheritance, signatures). As the syntax and semantics of F-logic are much more powerful than that of Datalog, no suitable notion of unification exists so that the strategy of semi-naive evaluation is not applicable in a straightforward way. This paper investigates the difficulties arising and presents an algorithm for an incremental bottom-up evaluation of F-logic programs which has been implemented by the FLORID prototype.

1 Introduction

Compared with relational databases, a major advantage of deductive databases is possibility to define views *recursively.* A typic example is the computation of the transitive closure of a given relation :

$$\begin{aligned}&\mathsf{path(X,Y)} \mathrel{:-} \mathsf{edge(X,Y).}\\&\mathsf{path(X,Y)} \mathrel{:-} \mathsf{path(X,Z),\ edge(Z,Y).}\end{aligned} \tag{1.1}$$

First order logic is not sufficient for such queries, a fixpoint operator is needed to define a semantics. The simplest evaluation strategy is to apply the rules to all known facts, add the newly derived facts (T_P operator) and repeat this process until no new facts are derived anymore (fixpoint of T_P). This so-called *naive evaluation* has the disadvantage of deriving the same facts over and over. As the T_P operator of a program P (not containing negation) is monotone, each evaluation round yields all the facts inferred in earlier rounds plus some new ones. The old/new ratio may be very unfavourable especially with many T_P rounds. The basic idea of the semi-naive evaluation strategy is to avoid inferring a fact twice *in the same way.* In order to do this, when instanciating the rules to compute T_P, at least one subgoal of each rule has to be instanciated with a fact newly derived in the last evaluation round. Thus, for every IDB predicate p we have to deal with the set $\Delta^i p$ containing those tuples of p newly derived in round i. In our transitive closure program, we get

$$\mathsf{path}^{i}_{new}(\mathsf{X,Y}) := \Delta^{i-1}_{path}(\mathsf{X,Z}) \times \mathsf{edge}(\mathsf{Z,Y})$$

$$\begin{aligned} &\Delta^{i}_{path} := \mathsf{path}^{i}_{neu} \setminus \mathsf{path}^{i-1} \\ &\mathsf{path}^{i} := \mathsf{path}^{i-1} \cup \Delta^{i}_{path} \end{aligned} \tag{1.2}$$

This can be implemented by transforming the program rules into "delta rules" and ensuring that in each round i the predicates and delta predicates are properly set:

$$\mathsf{path}(\mathsf{X,Y}) := \Delta_{path}(\mathsf{X,Z}),\ \mathsf{edge}(\mathsf{Z,Y}) \tag{1.3}$$

In the general case (more than one recursive subgoal) a rule with n IDB subgoals gives rise to n delta rules. For a thorough description of semi-naive evaluation see [8].

The aim of this paper is to develop a semi-naive evaluation algorithm for F-logic programs. F-logic[7] is a powerful language designed to combine object oriented features with deduction. But the strengths of F-logic makes it difficult to apply semi-naive evaluation techniques. (For an introduction to the F-logic syntax and semantics, we refer to [6].)

The transformation of Datalog programs makes use of the possibility to define a dependency graph for Datalog programs: A subgoal depends from a rule head if both unify, essentially if the predicate names are identical. To each predicate name appearing in a rule head, a delta predicate is created. No dependency between predicates with different names can occur. The straightforward way to lift the semi-naive evaluation techique to F-logic would be to build a dependency graph between heads and subgoals of rules (this normally employs a notion of syntactic unification), to detect cycles (recursion) and use some kinds of delta constructs for these cycles.

It is difficult, though, to find a sufficient notion of unification for F-logic. When we try to decide syntactically whether a subgoal possibly depends from a rule head, several problems arise in F-logic.

- **Complex molecules.** Information concerning the same object may be bundled to a complex molecule. For example the fact john:man[age→24; kids↠{peter,paul}] is equivalent to the conjunction of the atoms john:man, john[age→24], john[kids↠peter], and john[kids↠Paul]. Thus, a unification notion for molecules has to be directed and consider questions of subsumption (cf. [7], chapter 11).

- **Variables ranging over methods.** As F-logic has a higher-order syntax (while its semantics is still first order), there is no syntactic distinction between classes, instances and methods. F-logic admits variables on nearly all positions (exept for ID-term constructors). Thus, a subgoal of the type X[Y→Z] may depend on any fact defining a scalar method without parameters. It is very common to model Datalog predicates by method calls in F-logic(e.g. father(mary,john) by mary[father→john]), but the syntactic dependency is lost by doing this (on the other hand, we gain the opportunity to ask meta queries not possible in Datalog).

- **Closure properties of the class hierarchy.** There are a number of axioms that must hold in every model of an F-logic program. Concerning the isa-relationship, namely we have *isa reflexivity* (o :: o holds for all objects o), *isa transitivity* (c :: d, d :: e entails c :: e) and *subclass inclusion* (o : c and c :: d implies o : d). These have to be considered when deciding dependency between isa facts. If we view ":" and "::" as infix predicates, they are interdependent. Moreover, subclass facts have to be considered as "new" (because they yield new facts when instanciating a rule subgoal) that have never been directly inferred. Therefore it is not enough just to save the newly derived isa and subclass tuples to delta predicates and transform the rules as usual.

- **Equality.** The built-in predicate "=" with the usual equality axioms *reflexivity, symmetry, transitivity* is integral part of the F-logic semantics (by defining a congruence relation on the ID-terms). Thus, ground atoms may depend on other, different ground atoms. As equating of ID-terms may happen dynamically, even implicitely (to avoid inconsistencies when defining a scalar method twice, e.g. thisbook[author→karol_woytila], thisbook[author→johannes_paul_II], or when resolving cycles in a hierarchy definition) so it cannot be taken into account staticly.
- **Inheritance.** Inheritance can be viewed as a special type of closure property. Inheritable methods defined on classes apply to the class members without explicitly stating. By this we have potential dependencies between method atoms of different arrow type. Inheritance can be overridden, though, by deduction. In contrast to the most other DOOD languages, F-logic imposes no restrictions on inheritance. To resolve inheritance conflicts, F-logic employs a non-deterministic *trigger semantics*, thus choosing one possible model of the program. The procedural aspects of this trigger semantics have to be taken into account when evaluating programs incrementally.

These difficulties make it impossible to evaluate an F-logic program incrementally by just analyzing dependencies and transforming the program analogously to the Datalog case.

One approach to cope with these problems would be to confine semi-naive evaluation only to programs from a subclass of F-logic where a reasonable unification is possible. Such a sublanguage is *hierarchy-static F-logic*, where no rule-based subclass definitions is allowed and equating of objects is considered as inconsistency. Hierarchy-static F-logic is investigated by Frohn [2] in order to define a magic-set algorithm. However, such a sublanguage seems to be too restricted as just the most interesting features of F-logic are lost. Our aim was to find an algorithm which is fast if program and EDB are apt for semi-naive evaluation (relatively simple-structured recursions with many T_P rounds and small sets of new facts derived in each round) but yields the correct model for all programs.

The rest of the paper is structured as follows. The next section explains our approaches to the problems mentioned above. Section 3 formally defines the algorithm resulting from these ideas and shows its correctness. Then, the program rewriting is demonstrated with a few examples. Section 5 deals with possible enhancements of the algorithm. Section 6 and 7 detail what was actually implemented in the context of the FLORID project [3] [4] [5] and its usage. How to use semi-naive evaluation for programs with negation is shortly explained in section 8. In the conclusion, we present some performance results and discuss the effect of this optimization in practice.

2 Basic Ideas

We divide the evaluation algorithm into two major parts handled differently. The first part is the *deduction of new facts* according to the program rules and the partial model calculated so far (in FLORID, this is the contents of the *object manager* (OM)). For evaluating semi-naively, we rewrite the program rules using some (yet to define) delta predicates that are assumed to contain new facts derived in the last round. In other words, this part of our strategy amounts to a purely syntactic program transformation. The second part is to *insert the facts into the partial model*. In the naive case, this is merely an insertion of some ground substitions of rule heads. For semi-naive evaluation, we additionally have to maintain the delta predicates. All predicates are cleared before doing the insert. While inserting, all new facts are inserted into the respective delta predicates, too (maybe in a different syntax). This second part of the algorithm can be handled separately (and on a lower level) from the first one as no information about the program is needed, only a set of delta predicates to maintain. Thus, the insert part of our strategy is described more detailed in the implementation part of this paper; the formal treatment focuses on the transformation.

Here, we address the problems mentioned in the following way:

- We suppose all subgoals to be atoms (or negated atoms), that is, only simple molecules of the form obj[meth$\rightsquigarrow$res], obj[meth@(arg$_1$,...,arg$_n$)$\rightsquigarrow$res], o : c, c :: d and predicate atoms are allowed ($\rightsquigarrow$ stands for any of the arrows $\rightarrow, \twoheadrightarrow, \Rightarrow, \Rrightarrow, \bullet\!\!\rightarrow, \bullet\!\!\twoheadrightarrow$). Complex molecules can be transformed into conjunctions of atoms (*flattening*). Resolving path expressions (john.spouse.age) requires the introduction of new variables (john[spouse$\rightarrow$Y], Y[age$\rightarrow$Z]); flattening negated complex molecules may even lead to new rules defining auxiliary predicates. All this is done in a *normalization step.*

- Disregarding inheritance for a moment, all dependency between data atoms can only involve atoms of the same *arrow/arity combination.* Since variables may occur on every position, this is the sharpest neccessary syntactic criterion possible. We therefore assign a delta predicate $\Delta^n_{\rightsquigarrow}$ of arity $n+2$ to each arrow/arity combination appearing in the program. When transforming a rule, a subgoal of this type is deltaified as $\Delta^n_{\rightsquigarrow}$(meth,obj,arg$_1$,...,arg$_n$,res).

- We use Δ_{isa} and Δ_{subcl} as delta predicates of arity 2 for : and ::. As they are interdependent, simply deltaifying o : c as Δ_{isa}(o,c) is not sufficient. So we employ additional subgoals to exploit the built-in closure properties of F-logic. The object o is a new instance of class c if o : d has been newly derived for a subclass d of c, or if it is instance of a class d that became a subclass of c in the last round. Therefore, we transform o : c into (Δ_{isa}(o,D) , D :: c) $\vee$ (Δ_{subcl}(C,D), o : C, D :: c) with newly introduced variables C,D. The disjunction gives rise to two delta rules belonging to this subgoal. If no rule infers subclass atoms, the second rule is not needed.

- After equating objects during evaluation, the tuples in the delta predicates remain correct. However, these sets of tuples may be too small. If one of the equated object is directly referred to by a subgoal, the properties of the other object must be considered as "new facts" for this subgoal. One way to cope with this is to put *all* properties of the equated objects into the respective delt predicates again. As equating is supposed to occur seldom in programs apt for incremental evaluation (it may even be forbidden except for special parts of the program explicitely defining the equivalence structure) we chose the simpler way to apply the (untransformed) operator T_P once if an equating has occurred. This ignores the delta predicates—thus not losing any information—and updates them appropriately.

- Inheritance has a nested fixpoint semantics in F-logic where deduction is given priority over inheritance to enable overriding (non-monotonic inheritance). The model is calculated by alternating deduction (i.e. fixpoint of T_P) and deriving *one* new fact by inheritance ("firing" of *inheritance triggers*). To combine this semantics with semi-naive evaluation, all we have to do is to insert the new fact from a trigger into the respective delta predicate and to continue with the evaluation of the transformed program.

3 Formal Treatment

3.1 Conventions

For the formal exposition and investigation of the algorithm, we restrict ourselves to positive rules containing no negated subgoals. Semi-naive evaluation cannot deal with negation of recursive subgoals in the Datalog case, either. This problem can be solved by stratification, so that only predicates from a previous stratum occur negated. This solution applies to F-logic as well. Negation in combination with inheritance is even more involved (cf. [7], Appendix B). Moreover, we leave signature atoms and equality out of consideration. How to deal with equating objects was already suggested in the last section. Signature expressions are handled in FLORID just like other data expressions, no type checking is done during evaluation[1]. So we may treat signature atoms analogously to atoms of other arrow types.

The program is assumed to be in normalized form, i.e. it contains no complex molecules or path expressions (this can be done as a preliminary transformation step). Rule heads may consist of conjunctions of molecules (which is semantically equivalent to multiple rules with identical body).

We start the evaluation of the program with an empty partial model. That means that all EDB facts have to be part of the program. Applying a program to a non-empty structure is discussed in the implementation part of this paper.

3.2 Rule Rewriting

To rewrite an F-logic program P, we need the for each n the following additional predicates of arity $n+3$:

$\Delta^n_{\rightarrow}$	if P contains a subgoal of the form $o[m@(a_1,\ldots,a_n)\rightarrow r]$,
$\Delta^n_{\twoheadrightarrow}$	if P contains a subgoal of the form $o[m@(a_1,\ldots,a_n)\twoheadrightarrow r]$,
$\Delta^n_{\bullet\rightarrow}$	if P contains a subgoal of the form $o[m@(a_1,\ldots,a_n)\bullet\!\!\rightarrow r]$,
$\Delta^n_{\bullet\twoheadrightarrow}$	if P contains a subgoal of the form $o[m@(a_1,\ldots,a_n)\bullet\!\!\twoheadrightarrow r]$.

For deltaifying isa and subclass atoms, we need the binary predicates Δ_{isa} and Δ_{subcl}. As in the Datalog case, for every predicate p occuring in the head (IDB predicate) a delta predicate Δ_p is required. The (finite) set of predicate symbols thus defined is assumed to be disjoint from the set of predicate symbols occuring in P and is denoted by $\mathcal{P}_\Delta$.

In the following we define a rewriting function Δ by start with atoms and then lifting the function to rule bodies, rules and programs. ΔP then is the rewritten program.

Definition 1 *For an atom A, let ΔA be the following:*

[1]However, it is possible to check a model for correct typing *after* it is established.

A	ΔA
$o[m@(a_1,\ldots,a_n \rightsquigarrow r]$	$\Delta^n_{\rightsquigarrow}(m,o,a_1,\ldots,a_n,r)$
$o :: c$	$o :: C' \wedge \Delta_{subcl}(C',C'') \wedge C'' :: c$
$o : c$	$(\Delta_{isa}(o,C') \wedge C' :: c) \vee (o : C', \Delta_{subcl}(C',C'') \wedge C'' :: C)$
$p(a_1,\ldots,a_n)$	$\Delta_p(a_1,\ldots,a_n)$, *if* p *is an IDB predicate*
	false, if p *is an EDB predicate.*

where C' *and* C'' *are new variable symbols, not occuring anywhere else in* P.
We lift Δ *to rule bodies* $b = \bigwedge_{i=1}^{n} A_i$ *by the equation*

$$\Delta(\bigwedge_{i=1}^{n} A_i) := DNF(\bigvee_{j=1}^{n}((\Delta A_j) \wedge \bigwedge_{i \neq j} A_i)).$$

where DNF *denotes the operation that maps arbitrary boolean terms onto equivalent terms in disjunctive normal form.*

Delta transformation of rules $r = h :- b$ *is declared as*

$$\Delta r = \{H :- B' | B' \text{ branch of the disjunction } \Delta B\}$$

Finally, the delta program ΔP *is defined as*

$$\Delta P = \bigcup \{\Delta r | r \in P\},$$

the set of all delta rules produced by rules of P.

By this definition, we get for each rule r a set of rules with identical head and exactly one delta goal in the body. If the body of r consists only of n IDB predicates, Δr will be the set of rules delivered by the standard semi-naive transformation technique. As EDB predicates are transformed into false, the respective rules may be omitted.

In general, a rule with k subgoals, of which i are of the form `o : c`, will yield $k + i$ delta rules. Note that the rule heads are left untransformed because the maintenance of the delta predicates is considered separately and does not affect program transformation.

3.3 Evaluating Inheritance-Free Programs

For a program not making use of inheritance, it is sufficient to evaluate the delta program defined in the previous section such that the predicates from $\mathcal{P}_\Delta$ are properly set up to the new tuples of each round. We sketch this algorithm in a PASCAL-like pseudo-code:

```
Rewrite Program;
clearDeltaPreds;
insert facts into partial model and delta predicates;
REPEAT
  FOR all rules of delta program DO
    Retrieve bindings yielded by rule body;
  clearDeltaPreds;
  Insert heads with bindings yielded by rule bodies and
    insert new facts into the respective delta predicates;
UNTIL (DeltaPreds empty);
```

Theorem 1 (Correctness for inheritance-free programs)
For inheritance-free programs in accordance with the conventions above, the presented semi-naive evaluation will yield the same model as naive evaluation. □

Proof
For a H-structure J for ΔP let $J|_P$ be the H-structure obtained by removing all facts regarding delta predicates. We have to verify that naive and semi-naive evaluation coincide, that is

$$T_P^\infty(\emptyset) = T_{\Delta P}^\infty(\emptyset)|_P.$$

We show $T_P^n(\emptyset) = T_{\Delta P}^n(\emptyset)|_P$ by induction over n:
$n = 1$: In the first evaluation round, in both cases exactly the facts from P are inserted. Therefore $T_P^1(\emptyset) = T_{\Delta p}^1(\emptyset)|_P$ holds.
$n \mapsto n+1$: $T_P^{n+1}(\emptyset) = T_P(T_P^n(\emptyset)) = T_P(T_{\Delta P}^n(\emptyset)|_P)$ according to induction hypothesis. We have to show that $T_P(T_{\Delta P}^n(\emptyset)|_P) = T_{\Delta P}(T_{\Delta P}^n(\emptyset))|_P$.
$\subset$: Let f be a fact derived by T_P. Case 1: $f \in T_P^n(\emptyset)$. By induction hypothesis, this implies $f \in T_{\Delta P}^n(\emptyset)|_P$ and thus $f \in T_{\Delta P}^{n+1}(\emptyset)$, because evaluation (besides the delta predicates) is monotone. Case 2: $f \notin T_P^n(\emptyset)$. Then there is a rule $h : -l_1, \ldots, l_k$ in P and a ground substitution σ such that $f = h\sigma$ and $l_j\sigma \in T_P^n(\emptyset)$ holds for all j. Because f is new, there has to be a j such that $l_j\sigma \notin T_P^{n-1}(\emptyset)$. From the program rewriting definitions, it is clear that $h : -l_1, \ldots, l_{j-1}, \Delta l_j, l_{j+1}, \ldots, l_k \in \Delta P$. We claim:

$$(*) \qquad (\Delta l_j)\sigma' \in T_{\Delta P}^n(\emptyset) \text{ for a ground substitution } \sigma' \supset \sigma.$$

If Δl_j is a boolean combination of atoms, we define $(\bigvee \bigwedge A_{ij})\sigma \in M$:iff $\bigvee \bigwedge (A_{ij}\sigma \in M)$.
From $(*)$ then follows $h\sigma (= h\sigma') \in T_{\Delta P}^{n+1}(\emptyset)$.
$\supset$: For $f \in T_{\Delta P}^{n+1}(\emptyset)$ there is a rule $h : -l_1, \ldots, l_{j-1}, \Delta l_j, l_{j+1}, \ldots, l_k \in \Delta P$ and a ground substitution σ, such that $f = h\sigma$. We have to show that $l_s\sigma \in T_P^n(\emptyset)$ holds for all $s, 1 \le s \le k$. This is trivial for all unchanged subgoals ($s \neq j$). For $(\Delta l_j)\sigma = \Delta_{\leadsto}^n(m, o, a_1, \ldots, a_s, r)$, the fact $l_j\sigma = o[m@(a_1, \ldots, a_s) \leadsto r]$ has been derived in the previous evaluation round. Hence, it is in $T_P^n(\emptyset)$. To show the claim for hierarchy facts, use the closure properties of F-logic together with the additional subgoals exploiting these.
It remains to show that (*) holds. If $l_j\sigma = o[m@(a_1, \ldots, a_s) \leadsto r]$ we choose $\sigma' = \sigma$. Due to the assumption we have $l_j\sigma = o[m@(a_1, \ldots, a_s) \leadsto r] \in T_P^n(\emptyset)$, but $\notin T_P^{n-1}(\emptyset)$. From induction hypothesis and the definition of the algorithm it follows that $l'_j\sigma \in \Delta_{\leadsto}^s$.
In the case of $l_j\sigma = o :: c$, this fact is new only if the class hierarchy has been changed in the previous round. Let $a :: b$ the new fact relevant for the rule in question, i.e. $(a, b) \in \Delta_{subcl}$. Choose $\sigma' := \sigma \bigcup \{C'/a, C''/b\}$. From the closure properties of F-logic we have $o :: a, \Delta_{subcl}(a, b)$ and $b :: c$ in $T_{\Delta P}^n(\emptyset)$, and hence $l'_j\sigma' \in T_{\Delta P}^n(\emptyset)$. The last case $l_j\sigma = o : c$ is shown analogously.
□

3.4 Evaluating Programs with Inheritance

To incorporate inheritance, we have to extend the formal notion of inheritance trigger from [7] and [9] by delta predicate maintainance.

Let $\tau = \langle obj : cl, m@a_1, \ldots, a_n, \bullet\!\!\leadsto \rangle$ be an active trigger. That means that $obj : cl$ holds in the partial model, the method m (with parameters $a_1, \ldots, a_n$) is defined on cl as inheritable and the atom $obj[m@\ldots\rightsquigarrow\ldots]$ matches some goal from P. Then we define the operator $\Delta\tau$ (firing the trigger) applied to an H-structure I as

$$\Delta t(I) = I \cup \{obj[m@a_1, \ldots, a_n \rightarrow v], \Delta^n_{\rightsquigarrow}(m, obj, a_1, \ldots, a_n) | v \in I(cl, m@a_1, \ldots, a_n, \bullet\!\!\rightsquigarrow)\}$$

This explicitely inserts the new fact into the appropriate delta predicate. Thus, inheritance can seamlessly be integrated into the semi-naive evaluation described so far.

Note that to an active trigger there is always a delta predicate in $\mathcal{P}_\Delta$ because of the matching subgoal in P.

The extended algorithm looks as follows:

```
Rewrite Program;
clearDeltaPreds;
insert facts into partial model and delta predicates;
REPEAT
  REPEAT
    FOR all rules of delta program DO
      Retrieve bindings yielded by rule body;
    clearDeltaPreds;
    Insert heads with bindings yielded by rule bodies and
      insert new facts into the respective delta predicates;
  UNTIL (DeltaPreds empty);
search active trigger in original program;
IF (active trigger)
 THEN realize trigger;
UNTIL (no active trigger found);
```

Note that for searching active triggers the original program is used. That is because there has to be a matching subgoal to activate the trigger. If this subgoal is deltaified, matching is lost, so the search algorithm will not find this trigger.

Theorem 2 (Correctness)
For all programs in accordance with the conventions above, semi-naive evaluation will yield the same model as naive evaluation. □

Proof
Let $\tau_1, \ldots, \tau_t$ be a sequence of triggers and $\emptyset = I_0, \ldots, I_{t+1}$ a sequence of H-structures for P where

(1) τ_1 is active trigger in $I_1 := T_P^\infty(\emptyset)$

(2) $\forall i > 1$ is τ_i active trigger in $I_i := T_P^\infty(\tau_{i-1}(I_{i-1}))$

(3) $I_{t+1} := T_P^\infty(\tau_t(I_t))$ has no active triggers.

Here $\tau(I)$ denotes firing the trigger τ, i.e. extending I by the new fact yielded by the trigger. This sequence describes the naive evaluation of P (cf. [7]). Then we have to show:
There is a sequence of H-structures $J_1, \ldots, J_{t+1}$ for ΔP such that

(α) $\forall i J_i|_P = I_i$

(β) τ_1 is active trigger (for P) in $T^{\infty}_{\Delta P}(\emptyset)$

(γ) $\forall i > 1$ is τ_i active trigger (for P) in $J_i := T^{\infty}_{\Delta P}(\Delta\tau_{i-1}(J_{i-1}))$

(δ) $J_{t+1} := T^{\infty}_{\Delta P}(\Delta\tau_t(J_t))$ has no active triggers.

We show the statements by induction over i:

$i = 1 : J_1 := T^{\infty}_{\Delta P}(\emptyset)$ Then (α) follows from the previous theorem. As triggers are not affected by removing predicate facts, (α) implies (β). (The last two statements do not apply here)

$i > 1$: We have to show that $J_i = T^{\infty}_{\Delta P}(\Delta\tau_{i-1}(J_{i-1}))$ equals to $I_i = T^{\infty}_{P}(\tau_{i-1}(I_{i-1}))$. From induction hypothesis (γ) and (α) we have $I_i = T^i_P nfty(\Delta\tau_{i-1}(J_{i-1}))$. Analogous to the proof of the previous theorem, we show $T^n_P(\tau_{i-1}(J_{i-1})) = T^n_{\Delta P}(\Delta\tau_{i-1}(J_{i-1}))$ by induction over n. For $n = 1$, any new fact derived by T_P must come from a rule that uses the trigger fact to match one of the subgoals (otherwise there is a contradiction to the fixpoint reached before). On the other hand, exactly this fact (in deltaified form) is contained in the delta predicates of J_i. Thus, we can argue like in the induction step of the previous proof. The induction step carries over literally (when the empty set is replaced by $\Delta\tau_{i-1}(J_{i-1})$). Hence, both fixpoints are equal and the statement (α) holds for i. As the triggers are not affected by restricting to P, this also yields (γ). Finally, (δ) is clear from (α) because an active trigger in J_{t+1} would be an active trigger in I_{t+1} in contrast to (3). $\Box$

4 Example Programs

To illustrate the program transformation, in this section we give some example programs together with their delta version. It is evident from the examples that normalization and rule rewriting may produce a fair overhead, especially if the rules get more complex. In the delta programs, we did some obvious optimizations: subgoals not affected by the delta transformation are left in complex form and delta predicates not affected by rule heads are omitted. This optimizations are incorporated into the implementation, too (see section 5). We list only the rules, as the facts are not affected by the transformation.

Example 1
Transitive closure using attributes

```
edge::path.

p(E,P):path[start->X; end->Z] :- E:edge[start->X; end->Y],
                                 P:path[start->Y; end->Z].
```

The corresponding delta program:

```
edge::path.

p(E,P):path[start->X; end->Z] :- delta_isa(E,C),C::edge,
                                 E[start->X; end->Y],
                                 P:path[start->Y; end->Z].
```

```
p(E,P):path[start->X; end->Z] :- delta_sv(1,E,X),E:edge[end->Y],
                                 P:path[start->Y; end->Z].

p(E,P):path[start->X; end->Z] :- delta_sv(end,E,Y),E:edge[start->X],
                                 P:path[start->Y; end->Z].

p(E,P):path[start->X; end->Z] :- delta_isa(P,C),C::path,
                                 E:edge[start->X; end->Y],
                                 P[start->Y; end->Z].

p(E,P):path[start->X; end->Z] :- delta_sv(1,P,Y),
                                 E:edge[start->X; end->Y],
                                 P:path[end->Z].

p(E,P):path[start->X; end->Z] :- delta_sv(end,P,Z),
                                 E:edge[start->X; end->Y],
                                 P:path[start->Y].
```

We need six delta rules for the single original rule because normalizations yields six subgoals. Therefore it is more efficient to model such simple problems in datalog. This is not always possible, though, as the next example shows: □

Example 2
Transitive closure using class hierarchy closure axioms.

```
horse::animal.
mare::horse.
stallion::horse.

fury:stallion.

Y:mare :- X:horse, X[mother->Y].
Y:stallion :- X:horse, X[father->Y].
```

Here, the dependancy between heads and rule bodies is not staticly decidable (without using the EDB). Transformation yields:

```
Y:mare :- delta_isa(X,Z), Z::horse, X[mother->Y].
Y:stallion :- delta_isa(X,Z), Z::horse, X[father->Y].
```

This example clearly shows the role of the additional subgoals encoding the closure properties. □

Example 3
The following program calculates the possible intervals between the tones of an octave and uses inheritance.

```
/* basic tones*/


c:tone[number->1;chro->1].
d:tone[number->2;chro->3].
e:tone[number->3;chro->5].
f:tone[number->4;chro->6].
g:tone[number->5;chro->8].
a:tone[number->6;chro->10].
b:tone[number->7;chro->12].

tone::note.

/* basic intervals */

prime::interval[diff*->0].
second::interval[diff*->1].
third::interval[diff*->2].
fourth::interval[diff*->3].
fifth::interval[diff*->4].
sixth::interval[diff*->5].
seventh::interval[diff*->6].

/* derived intervals */

plainPrime:prime[chrodiff->0].

dimSecond:second[chrodiff->0].
smallSecond:second[chrodiff->1].
bigSecond:second[chrodiff->2].

dimThird:third[chrodiff->2].
smallThird:third[chrodiff->3].
bigThird:third[chrodiff->4].
augThird:third[chrodiff->5].

dimFourth:fourth[chrodiff->4].
plainFourth:fourth[chrodiff->5].
augFourth:fourth[chrodiff->6].

plainFifth:fifth[chrodiff->7].
dimFifth:fifth[chrodiff->6].
augFifth:fifth[chrodiff->8].

dimSixth:sixth[chrodiff->7].
smallSixth:sixth[chrodiff->8].
bigSixth:sixth[chrodiff->9].
```

```
augSixth:sixth[chrodiff->10].

dimSeventh:seventh[chrodiff->9].
smallSeventh:seventh[chrodiff->10].
bigSeventh:seventh[chrodiff->11].

/* rules for deriving alterations */

sharp(X):note[number->X.number;chro->X.chro + 1]:-X:tone.
flat(X):note[number->X.number;chro->X.chro - 1]:-X:tone.


/* rule for calculating intervals */

X[to@(Y)->Z] :- X:note,Y:note,
                V=Y.chro - X.chro, U=Y.number - X.number,
                Z:interval[diff->U;chrodiff->V].
```

Normalizing and rewriting yields

```
sharp(X):note[number->X.number;chro->X.chro + 1]:- delta_isa(X,C), C::tone.
flat(X):note[number->X.number;chro->X.chro - 1]:- delta_isa(X,C), C::tone.


X[to@(Y)->Z] :- delta_isa(X:C), C::note,
                Y:note[chro->CY; number->NY]
                X[chro->CX; number->NX], V=CY-CX, U=NY-NX,
                Z:interval[diff->U; chrodiff->V].

X[to@(Y)->Z] :- delta_isa(Y:C), C::note,
                Y[chro->CY; number->NY]
                X:note[chro->CX; number->NX], V=CY-CX, U=NY-NX,
                Z:interval[diff->U; chrodiff->V].

X[to@(Y)->Z] :- delta_sv(chro,Y,CY),
                Y:note[number->NY]
                X:note[chro->CX; number->NX], V=CY-CX, U=NY-NX,
                Z:interval[diff->U; chrodiff->V].

X[to@(Y)->Z] :- delta_sv(chro,X,CX),
                Y:note[chro->CY; number->NY]
                X:note[number->NX], V=CY-CX, U=NY-NX,
                Z:interval[diff->U; chrodiff->V].

X[to@(Y)->Z] :- delta_sv(number,Y,NY),
                Y:note[chro->CY]
                X:note[chro->CX; number->NX], V=CY-CX, U=NY-NX,
```

```
                  Z:interval[diff->U; chrodiff->V].

X[to@(Y)->Z] :- delta_sv(number,X,NX),
                  Y:note[chro->CY; number->NY]
                  X:note[chro->CX], V=CY-CX, U=NY-NX,
                  Z:interval[diff->U; chrodiff->V].

X[to@(Y)->Z] :- delta_isa(Z,C),C::interval,
                  Y:note[chro->CY; number->NY]
                  X:note[chro->CX; number->NX], V=CY-CX, U=NY-NX,
                  Z[diff->U; chrodiff->V].

X[to@(Y)->Z] :- delta_sv(diff,Z,U),
                  Y:note[chro->CY; number->NY]
                  X:note[chro->CX; number->NX], V=CY-CX, U=NY-NX,
                  Z:interval[chrodiff->V].

X[to@(Y)->Z] :- delta_sv(chrodiff,Z,V),
                  Y:note[chro->CY; number->NY]
                  X:note[chro->CX; number->NX], V=CY-CX, U=NY-NX,
                  Z:interval[diff->U].
```

The complex interval rule gives rise to many delta rules. But not all of them are "active" at the same time, i.e. the delta subgoal is not empty. The time-consuming factor in this program is inheritance triggering. There are 21 active triggers, leading to 21 deductive fix points. After inserting the trigger fact, however, only the second last rule is active. For fast detection of inactive rules, we always position the delta subgoal in the first place. If we omit delta predicates that are not needed in the body of a rule, we are even able to detect the next fixpoint after *one* application of $T_{\Delta P}$ because all delta predicates are empty.
Dividing the program into two parts (by user stratification before the last rule) and thus separating inheritance and class hierarchy changes from the complex rule makes the transformation even more efficient. □

5 Optimizations

The semi-naive algorithm as presented is very general and has the advantage of coping with the class of all F-logic programs. Furthermore, it was implementable into the FLORID prototype with a reasonable effort.

There are several possibilities to further optimize the algorithm. As we mentioned in section 4, complex subgoals may be left complex if they are not affected by the delta transformation. Thus, normalization is only needed *locally*, on a subgoal level. As complex molecules have more information (for execution) than the equivalent conjunction of atoms, this information is not lost unnecessarily.

Further improvements may use the real dependancies between heads and subgoals. The simplest step is to remove from $\mathcal{P}_\Delta$ all predicates which are not needed to rewrite rule bodies. For restricted subclasses of F-logic we can do more. If all method calls are ground (no variables on method position), for each method name (and each arrow/arity combination) a special delta predicate can be used. This has been actually implemented, see section 6.

Programs that do not dynamically extend the class hierarchy (this is checkable by a program analysis: no subclass atoms in rule heads, and is the case for most programs) allow a simpler deltaifying of isa subgoals:

A	$\Delta(A)$
$o[m@(a_1,\ldots,a_n{\leadsto}r]$	$\Delta^n_{\leadsto}(m,o,a_1,\ldots,a_n,r)$
$o :: c$	false
$o : c$	$\Delta_{isa}(o,C') \wedge C' :: c$
$p(a_1,\ldots,a_n)$	$\Delta_p(a_1,\ldots,a_n)$

This also reduces the number of delta rules.

In the class of hierarchy-static F-logic programs (no equating of objects, no rule-based hierarchy definitions, class hierarchy known to static analysis) we are able to build a dependancy graph. Assigning delta predicates to each interesting edge in this graph keeps the number of tuples in these predicates as small as possible. As no subclass atoms may be derived and we do not need to consider inheritance dependancies, all interesting edges are of the following types:

- Two method calls of the same type
- Two isa atoms
- Two predicates atoms with the same predicate name

When inserting ground substituted heads, the Object Manager (OM) of FLORID has to know which delta predicates must be updated. If this depends not only on the syntactical structure of the fact but on the rule, too, a quite complicated information exchange between transformation and insert part of the evaluation algorithm is necessary. Because this does not fit easily into the hierarchical architecture of FLORID, this optimization was not implemented yet.

6 Implementation Details

To incorporate the semi-naive evaluation algorithm into the FLORID prototype, some special cases had to be taken into account. This section describes the implemented algorithm, tells how to use this evaluation mode and gives some details about the program code. For more information about FLORID please see the User Manual [5].

6.1 Variants of Semi-Naive Evaluation

At the moment, the implemented algorithm knows five cases of rewriting which apply under different circumstances. More than one may occur in the same program.

1. **Predicates.** This is the standard Datalog situation. Assign a delta predicate Δ_p to a predicate symbol p that occurs in a rule head and as a subgoal.

2. **Data molecules.** To data molecules of the form o[m@$a_1,\ldots,a_n$)$\leadsto$r] assign delta predicates $\Delta^{arity}_{\leadsto}(m,o,a_1,\ldots,a_n,r)$ according to the arrow/arity combination. Putting the

method name in the first position is meant for quick access of the data structure for predicate frames (left to right nested trees).

3. **Ground method calls.** If for an arrow/arity combination all method calls in a program are ground, we may use a specific delta predicate for each method name in this combination. So the molecule o[meth@$a_1, \ldots, a_n$)$\rightsquigarrow$r] is transformed to $\Delta^n_{\rightsquigarrow,meth}(o, a_1, \ldots, a_n, r)$. When this optimization is used, no equating of method names must occur. It is possible to check on run-time whether method names used for delta predicates are affected by equatings. However, this is not implemented yet.

4. **Static class hierarchy.** If there are no heads of the form c :: d in the program, the class hierarchy is static. Isa atoms can then be transformed in a simpler way, see section 5.

5. **Empty sets and signatures.** Data molecules with multivalued methods and non-empty sets at the result position are equivalent to a conjunction of atoms with exactly one result, e.g. o[m$\twoheadrightarrow${r,s}] is equivalent to o[m$\twoheadrightarrow$r], o[m$\twoheadrightarrow$s]. However, this is not true for empty sets. While empty conjunction would be tautologic, a subgoal o[m$\twoheadrightarrow${}] yields *false* if no fact was derived for this method and true if any fact was derived (o[m$\twoheadrightarrow${}] may be derived explicitely). See the FLORID tutorial [6] for details. The same is true for the case of empty signatures. So we need special delta predicates $\Delta^{\emptyset}_{\rightsquigarrow}(o, m)$ if a rule bodies makes use of empty sets or signatures. They are set the first time any fact about o and m is derived.

Compared to the theoretical treatment above, some additional points have to be considered in the FLORID environment:

1. **Normalizing of programs.** Normalizing a program as a whole can be achieved by using an instance of the class `Normalizer`. But as mentioned before, it is more effective to leave as much molecules as possible in complex form because less joins and less additional variables (in the case of path expressions) are needed. So we use the completely normalized program only for deciding the sets of delta predicates needed, and do normalizing before rewriting in a sloppy way: All negatively used complex molecules and path expressions are flattened (because this may lead to additional rules and cannot be handled locally) before the transformation, and while rewriting, only the molecule requested for deltaifying.

2. **Facts.** It is often the case that the object model is not empty when starting the evaluation (applying the program to an EDB). To set up the delta predicates correctly, the only possibility is to do the first deduction round with the *original* program and then insert the new facts into the delta predicates. For all subsequent rounds the delta program is used.

3. **Equatings.** By equating two objects, every property of one of them has to be considered as new for the other, even if it only appears in the result position of some method call on a third object. To avoid a complete scan of the object model, this problem is handled like the case of non-empty object models: the original program is used for the next round. This ensures correct evaluation but has the disadvantage that the algorithm degenerates to the naive algorithm on transitive closures which equate objects in every round.

4. **Inheritance triggers.** Three conditions make an active inheritance trigger: (1) an object o has to be instance (resp. direct subclass) of a class c, (2) for c is a inheritable method m defined, and (3) o[m*⇝.] must match with a subgoal of the program. Condition (1) and (2) are not accessible to syntactic analysis because the object model may be non-empty before starting the evaluation. For determining the possible triggers (and thus the delta predicates needed for inheritance) only the coarse condition (3) remains. To keep the number of predicates small, we use *two* different sets of delta predicates, one for the deduction process and one for inheritance. See the next section also.

5. **Active objects.** To enable safety-checking of rule bodies, in FLORID the subclass reflexivity axiom has been restricted to *active objects*, i.e. objects that occurred in rule heads. Consider the following program:

   dog[legs•→4].
   p(X) :- dog :: X. (6.1)

   The predicate p shall be true for all superclasses of dog. Thus p(dog) holds, because dog :: dog. That means that introducing the fact about dogs alters the class hierarchy in a hidden way. To cope with this, when introducing a new object o we have to insert $\Delta_{subcl}(o, o)$, too, if some subgoal refers to the class hierarchy.

6.2 The Implemented Algorithm

To link the two parts of the semi-naive evaluation algorithm (rewriting and insert), informations about the set of delta predicates $\mathcal{P}_\Delta$ have to be passed to the Object Manager (OM) of FLORID. For this task a data structure `delta` has been defined. It contains for each arrow type dictionaries that map arities to names of delta predicates. For ground method optimization, method names are mapped to delta predicate names. Using this structure, the OM can decide whether to insert a given fact into a delta predicate and quickly find the right one.

To have the least possible number of delta rules, two different sets of delta predicates are used. The first (`tpDelta`) is for deduction and the second (`triggerDelta`) for firing inheritance triggers. For the first, we only need predicates for arrow/arity combination that occur in rule heads *and* bodies, for the second, we only need delta predicates for *data atoms*, not for isa or subclass atoms or predicates. Two different delta sets lead to two different delta programs, `deltaProg` and `triggerProg` which must be used in the appropriate situation. To start with non-empty object models and to set up the delta predicates after equating of objects, the original program is evaluated once. Thus in the implemented algorithm we deal with three versions of the program for deduction and two delta predicate sets for insertion. The refined algorithm looks as follows:

```
Normalize program;

tpDelta:= Set of delta predicates for recursive goals;
deltaProg:= rewrite program using tpDelta;

triggerDelta:= Set of delta predicates possibly needed for firing triggers;
triggerProg:= rewrite program using triggerDelta;
```

```
break:= false;
REPEAT
  REPEAT
    IF ((first TP round) or (equating occurred))
    THEN
      FOR all rules of original program DO
        Retrieve bindings yielded by rule body;
    ELSE
      FOR all rules of deltaProg DO
        Retrieve bindings yielded by rule body;
    clearDeltaPreds(tpDelta);
    Insert heads with bindings yielded by rule bodies and
      insert new facts into the predicates of tpDelta;
  UNTIL (all predicates of tpDelta are empty);
  IF (no active trigger in original program)
  THEN break:=true;
  ELSE
    clearDeltaPreds(triggerDelta);
    fire trigger and insert new fact into predicates of triggerDelta;
UNTIL (break);
```

7 Using Semi-Naive Evaluation with FLORID

The naive evaluation component `NaiveEval` has been exchanged by a derivation `PrSemiNaive`. Besides the usual system commands `evaluate, retrieve, debugOn, debugOff` and `help` (see the User Manual [5]), `PrSemiNaive` understands the new command `mode` to switch between different evaluation modes. Possible parameters are:

`?- sys.theEval.mode@("naive")[].` Bypass the semi-naive evaluation. In this mode, `PrSemiNaive` acts just like `NaiveEval`.

`?- sys.theEval.mode@("seminaive")[].` Set the evaluation to the most general semi-naive algorithm. This works for all programs except for programs with unstratified negation.

`?- sys.theEval.mode@("ground")[].` When all method calls for an arrow/arity type are ground, the rewriting uses special delta predicates for each method name. Note that equatings between those method name must not occur because then the model is not properly calculated. Such programs have to be evaluated in "seminaive" mode.

Additionally, when in debug mode, the rewritten programs are printed. This also enables an estimation of the evaluation overhead caused by the semi-naive transformation.

8 Semi-Naive Evaluation and Negation

Handling of negated subgoals is a fundamental problem of semi-naive evaluation algorithms. A delta transformation of negated subgoals does not make sense because it is not possible to conclude from $\neg\Delta_p$ to $\Delta_{\neg p}$. Hence negated subgoals are never deltaified. If a program is stratified so that no cycle of the dependancy graph has a negated edge, all negated subgoals can be considered as EDB subgoals and no problem arises. In F-logic, the lack of a suitable dependancy graph makes this notion of stratification much to restrictive. In most cases of programs using negation, the correct evaluation order has to be enforced by external means (user stratification) or by additional subgoals that make a rule active only after all facts for the negated literals are established. It is the programmer's responsibility to ensure this. Then, semi-naive evaluation will work correctly.

9 Conclusion

The presented algorithm extends the well-known semi-naive evaluation algorithm to the needs of F-logic and the F-logic prototype FLORID. It guarantees for all programs (with a clean negation handling) the same semantics as with the naive evaluation. The algorithm also incorporates the standard (Datalog) semi-naive evaluation as a special case.

A number of benchmark test showed as expected that the benefit of the presented algorithm strongly depends on the program to evaluate and the number of TP rounds needed for the given EDB. If the program has simple transitive closures similar to Datalog, it will benefit most of the optimization. The following example defines the method anc as transitive closure over father:

$$\begin{array}{l} \mathsf{X[anc} \twoheadrightarrow \mathsf{Y]} \text{ :- } \mathsf{X[father} \twoheadrightarrow \mathsf{Y]}. \\ \mathsf{X[anc} \twoheadrightarrow \mathsf{Y]} \text{ :- } \mathsf{X[anc} \twoheadrightarrow \mathsf{Z]},\ \mathsf{Z[father} \twoheadrightarrow \mathsf{Y]}. \end{array} \tag{9.1}$$

Given a linear order of length n as EDB ($\{i\texttt{[father}\twoheadrightarrow i+1\texttt{]}|0 \leq i < n\}$), on a SPARCstation 4 we measured the following numbers (real time):

n	naive	semi-naive	ground opt
10	0.31sec	0.17sec	0.14sec
20	2.53sec	0.49sec	0.43sec
30	9.53sec	1.03sec	0.93sec
40	25.12sec	1.81sec	1.67sec
50	56.19sec	2.88sec	2.72sec
60	1min48.36sec	4.29sec	4.04sec
70	3min1.25sec	5.83sec	5.65sec
100	12min45.96sec	13.93sec	13.24sec

A binary tree of depth n shows less difference between naive and semi-naive evaluation:

n	naive	semi-naive	ground opt
5	0.52sec	0.13sec	0.14sec
6	2.70sec	0.44sec	0.42sec
7	12.84sec	1.58sec	1.56sec
8	59.81sec	6.79sec	6.51sec
9	4min42.23sec	27.96sec	26.97sec
10	23min30.74sec	2min06.21sec	2min02.37sec

The more complex the programs are, the more use the make of the powerful features of F-logic, the greater the overhead of semi-naive evaluation. Many non-EDB subgoals lead to many delta rules. The plain example of section 4 is for n=100 'only' eight times as fast if evaluated semi-naive. As in the Datalog case, programs with many non-EDB subgoals may even lead to a *worse* performance compared to the naive evaluation. This problem was investigated by Benzschawel [1] (Chapter 7). He gives an improved algorithm, called *quarternaive* *evaluation*, that in every case is faster than naive evaluation. The idea is to isolate common subexpressions of the delta rules by using distribution laws and then calculating them incrementally, with delta subexpressions. It would be interesting to incorporate this idea into the F-logic semi-naive algorithm in order to improve the worst-case performance. However, an implementation in the FLORID context would be quite an effort because the object algebra has to be extended to subgoal matrices with delta versions that are passed to later evaluation rounds.

In practice, a program that needs many TP rounds will probably run faster if evaluated semi-naively. For example, a program simulating the alternating bit protocol in F-logic (it has more than 100 lines and needs 243 TP rounds[2]) is sped up approximately by the factor three.

References

[1] S. Benzschawel. *Implementierung einer effizienten Anfrageauswertung für ein deduktives Datenbanksystem.* PhD thesis, Univ. Trier, 1995.

[2] J. Frohn. Statische Magic-Set Transformation von F-Logik Programmen. Manuskript.

[3] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID – Ein Prototyp für F–Logik. In K. Dittrich, editor, *Datenbanksysteme in Büro, Technik und Wissenschaft*, Informatik aktuell. Springer, 1997.

[4] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID: A prototype for F–Logic. In *Proc. Intl. Conference on Data Engineering*, 1997.

[5] J. Frohn, R. Himmeröder, P.-T. Kandzia, and C. Schlepphorst. FLORID, Version 1.0, User Manual, 1996. Available from ftp://ftp.informatik.uni-freiburg.de/pub/florid/manual.ps.gz.

[6] J. Frohn, R. Himmeröder, P.-T. Kandzia, and C. Schlepphorst. How to write F-Logic programs in FLORID, 1996. Available from ftp://ftp.informatik.uni-freiburg.de/pub/florid/tutorial.ps.gz.

[7] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[8] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, New York, 1988.

[9] T. Westmann. Ein Parser für erweiterte F-Logik. Studienarbeit Universität Mannheim, 1994.

[2] The program abp.flp is available from our WWW server: http://www.informatik.uni-freiburg.de/~dbis/flsys/morexamples.html