# FLORA: Implementing an Efficient DOOD System Using a Tabling Logic Engine*

Guizhen Yang and Michael Kifer

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794, U.S.A.
{guizyang, kifer}@CS.SunySB.EDU

**Abstract.** This paper reports on the design and implementation of FLORA — a powerful DOOD system that incorporates the features of F-logic, HiLog, and Transaction Logic. FLORA is implemented by translation into XSB, a tabling logic engine that is known for its efficiency and is the only known system that extends the power of Prolog with an equivalent of the Magic Sets style optimization, the well-founded semantics for negation, and many other important features. We discuss the features of XSB that help our effort as well as the areas where it falls short of what is needed. We then describe our solutions and optimization techniques that address these problems and make FLORA much more efficient than other known DOOD systems based on F-logic.

## 1 Introduction

Deductive object-oriented databases (abbr. DOOD) attracted much attention in early 1990's but difficulties in realizing these ideas and performance problems had dampened the initial enthusiasm. Nevertheless, the second half of the last decade witnessed several experimental systems [34, 20, 2, 24, 17, 27]. They, along with the proliferation of the Web and many recent developments, such as the RDF[1] standard, have fueled renewed interest in DOOD systems; in particular, systems for logic-based processing of object-oriented meta-data [15, 18, 28, 4, 5]. Also, a new field — processing of semistructured data — is emerging to address a specialized segment of the research on DOOD systems [1].

In this paper, we report our work on FLORA, a *practical* DOOD system that has already been successfully used to build a number of sophisticated Web-based information systems, as reported in [13, 19, 26]. By "practical" we mean a DOOD system that has high expressive power, is built on strong theoretical foundations and offers competitive performance and convenient software development environment.

* Work supported in part by a grant from New York State through the program for Strategic Partnership for Industrial Resurgence, by XSB, Inc., through the NSF SBIR Award 9960485, and by NSF grant INT9809945.

[1] http://www.w3.org/RDF/

FLORA is based on F-logic [22], HiLog [11], and Transaction Logic [8, 6, 9], which are all incorporated into a single, coherent logic language along the lines described in [22, 21]. However, rather than developing our own deductive engine for F-logic (such as the ones developed for FLORID [17, 27] or SiLRI [15]), we chose to utilize an existing engine, XSB [29], and implement FLORA through source-level translation to XSB. Apart from the benefits of saving considerable amount of time, our choice of XSB was motivated by the following considerations:

1. XSB augments OLD-resolution [32] with *tabling*, which extends the well-known Magic Sets method [3], thereby offering both goal-driven top-down evaluation and data-driven bottom-up evaluation [31].
2. Mapping of F-logic and HiLog into predicate calculus is well known [22, 11].
3. XSB is known to be an order of magnitude faster than other similar logic systems, such as LDL and CORAL [29].
4. XSB has compile-time optimizations particularly suited for source-level translation, such as specialization [30], unification factoring [14], and trie-based indexing (which permits indexing on multiple arguments of a predicate).

To the best of our knowledge, the first functioning F-logic prototype based on the source-level translation approach was FLIP [25]. FLIP served as the starting point and the inspiration for our own work. Fortunately, there was plenty of work left for us to do, because FLIP's translation was essentially identical to that described in [22] and it was rather naively relying on the ability of XSB to apply the right optimizations. As a result, the implementation of FLIP suffered from a number of serious problems. In particular:

1. As a compiler optimization, XSB's specialization does not apply to many programs obtained from a direct translation of F-logic [22]. This is even more so when HiLog terms (which FLIP did not have) occur in the program.
2. Although fundamental to evaluating F-logic programs, tabling cannot be used without discretion. First, tabling can, in some cases, cause unnecessary overhead. Second, tabling and databases updates do not work well together.
3. FLIP did not have a consistent object model and had limited support for path expressions, functional attributes, and meta-programming.
4. Finally, FLIP did not provide any module system, which basically confined users to a single program file, making serious software development difficult.

In this paper we discuss how these problems are resolved in FLORA. The full paper will present performance results, which compare FLORA with other systems that implement F-logic.

## 2 Preliminaries

In this section we review the technical foundations of FLORA — F-logic [22], HiLog [11], and Transaction Logic [8, 7] — and describe their naive translation using "wrapper" predicates. This discussion forms the basis for understanding the architecture of FLORA and the optimizations built into it.

### 2.1 F-logic

F-logic subsumes predicate calculus while both its syntax and semantics are still defined in object-oriented terms. On the other hand, much of F-logic can be viewed as a syntactic variant of classical logic, which makes implementation through source-level translation possible.

**Basic Syntax.** F-logic uses Prolog *ground* (*i.e.*, variable-free) terms to represent object identities (abbr., oid's), *e.g.*, john and father(mary). Objects can have scalar (single-valued), multivalued, or Boolean attributes, for instance,

```
mary[spouse→john, children↠{alice, nancy}].
mary[children↠{jack}; married].
```

Here spouse→john says that mary has a scalar attribute spouse, whose value is the oid john; children↠{alice, nancy} says that the value of the multivalued attribute children is a set that *contains* two oid's: alice and nancy. We emphasize "contains" because sets do not need to be specified all at once. For instance, the second fact above says that mary has one other child, jack. The attribute married in the second fact is Boolean: its value is *true* in the above example.

While some attributes of an object can be specified explicitly as facts, other attributes can be defined using inference rules. For instance, we can derive john[children↠{alice, nancy, jack}] with the help of the following rule:

$$\mathsf{X[children{\twoheadrightarrow}\{C\}]} :- \mathsf{Y[spouse{\rightarrow}X,\ children{\twoheadrightarrow}\{C\}]}. \tag{1}$$

Here we adopt the usual Prolog convention that capitalized symbols denote variables, while symbols beginning with a lower case letter denote constants.

F-logic objects can also have *methods*, *i.e.*, functions that return a value or a set of values when appropriate arguments are provided. For instance,

```
john[grade@(cs305,f99)→100, courses@(f99)↠{cs305, cs306}].
```

says that john has a scalar method, grade, whose value on the arguments cs305 and f99 is 100, and a multivalued method courses, whose value on the argument f99 is a set of oid's that contains cs305 and cs306. As attributes, methods can also be defined using rules.

One might wonder about the purpose of the "@"-sign in method specification. Indeed, why not write grade(cs305,f99) instead? The purpose is to enable meta-programming without using meta-logic. The "@"-sign trick makes methods into objects so that variables can range over them. For instance, the following rules

$$\begin{aligned} \mathsf{X[methods{\twoheadrightarrow}\{M\}]} &:- \mathsf{X[M@(\_){\rightarrow}\_]}. \\ \mathsf{X[methods{\twoheadrightarrow}\{M\}]} &:- \mathsf{X[M@(\_,\_){\rightarrow}\_]}. \end{aligned} \tag{2}$$

where the symbol "_" denotes a new unique variable, define a new method, methods, which for any given object collects those of the object's methods that take one or two arguments.

Thus, the "@"-sign is just a syntactic gimmick that permits F-logic to stay within the boundary of first-order logic syntax and avoids having to deal with terms like M(X,Y), where M is a variable. However, there is a better gimmick, HiLog [11], which will be discussed shortly.

Finally, we note that F-logic can specify *class membership* (*e.g.*, john : student), *subclass relationship* (*e.g.*, student :: person), *types* (*e.g.*, person[name⇒string]), and many other things that are peripheral to the subject of this paper.

**Translation into Predicate Calculus.** A general translation technique, called *flattening*, was described in [22]. It used a small, fixed assortment of *wrapper predicates* to encode different types of specifications. For instance, the scalar attribute specification mary[age→30] is encoded as fd(age,mary,[ ],30) whereas the multivalued method specification john[courses@(f99)↠{cs305, cs306}] is encoded as mvd(courses,john,[f99],cs305) ∧ mvd(courses,john,[f99],cs306).

However, one problem is that the indexing advantage is lost due to the small number of wrapper predicates used, since most Prolog systems index on predicate names. At first thought, one might think that the problem can be easily avoided if the encoding used method and attribute names as predicates instead of the "faceless" general wrappers. However, this is not the case, because variables are allowed to occur in place of method names, which would make the translated program second-order.

Recursion presents another serious difficulty. The naive translation scheme will most likely produce rules that are highly recursive, due to the small number of wrapper predicates used. For instance, consider the rule (1) presented earlier; its naive translation is as follows:

$$\mathsf{mvd(children,X,[\,],C)} :- \mathsf{fd(spouse,Y,[\,],X)},\ \mathsf{mvd(children,Y,[\,],C)}.$$

In general, evaluating such rules using a regular Prolog-style engine will go to infinite loop even if logically there is only a finite number of possible answers. In contrast, such rules present no problems to a tabling logic engine, like XSB, which uses memorization to terminate unnecessary loops in the evaluation.

For completeness, we note that class membership has its own translation, *e.g.*, isa(john,student), and so does the subclass relationship, *e.g.*, subclass(student,person). Type specifications have their own translation as well. In addition, a set of axioms must be added to enforce various properties of F-logic. For instance, we have to ensure that scalar attributes yield at most one value for any given object, that the subclass relationship is transitively closed, and that subclass membership is contained in the superclass membership.

Last but not least, although the non-monotonic part of F-logic— inheritance — cannot be directly translated into predicate calculus, it can still be encoded using Prolog-style rules and computed using XSB's efficient implementation of the *well-founded semantics* for negation [33].

### 2.2 HiLog

We have seen that one can do certain amount of meta-programming in F-logic, mostly owing to the "@"-sign gimmick. Although the rules in (2) show that

all method names can be collected using this trick, it is not easy to collect all method invocations (*i.e.*, methods plus their arguments). Our experience with FLORA 1.0 also shows that it is very convenient to treat both method names *and* method invocations uniformly as objects, because the "@"-sign trick is error-prone: people tend to forget to write down the "@"-sign (in F-logic, grade@(cs305,f99) is different from grade(cs305,f99)).

Fortunately, with the extension of HiLog [11], all these problems disappear. We illustrate HiLog through examples. The simplest yet most unusual one is the definition of the standard Prolog meta-predicate, call: call(X) :− X. This means that HiLog does not distinguish between function terms and atomic formulas: the same variable can range over both. Variables can also range over function symbols, as in X(Y,a). A query of the form ?− p(X), X, X(Y,X) is well within the boundaries of HiLog. The syntax for HiLog terms also extends that of classical logic. For instance, g(X)(f(a,X),Y)(b,Y) is perfectly fine. Of course, such powerful syntax should be used sparingly, but people have found many important uses for these features (see [11] for some).

Obviously HiLog is a suitable replacement for the "@"-sign gimmick. Now with the HiLog extension, users can write, say,

$$X[\mathsf{methods}\twoheadrightarrow\{M\}] :- X[M(\_,\_)\rightarrow\_]$$

instead of the rules shown earlier in (2). Trivial as it might appear, HiLog completely eliminates the need for special meta-syntax used in FLORA 1.0, and reduces the danger of programming mistakes. In addition, the underlying conceptual object model becomes much more consistent. The HiLog extension is implemented in the upcoming FLORA 2.0. Section 4 discusses the techniques that were developed to optimize the translation.

**Encoding in Predicate Calculus.** It turns out that the semantics of HiLog is inherently first-order and that it can actually be encoded using standard predicate calculus [11]. Although the translation is rather subtle, it is defined with just two recursive transformation functions (we omit steps irrelevant to the main subject): $\mathsf{encode}_a$, for translating formulas, and $\mathsf{encode}_t$, for translating terms:

1. $\mathsf{encode}_t(X) = X$, for each variable X.
2. $\mathsf{encode}_t(s) = s$, for each function symbol s.
3. $\mathsf{encode}_t(t(t_1,\ldots,t_n)) = \mathsf{apply}_{n+1}(\mathsf{encode}_t(t), \mathsf{encode}_t(t_1),\ldots, \mathsf{encode}_t(t_n))$.
4. $\mathsf{encode}_a(A) = \mathsf{call}(\mathsf{encode}_t(A))$, where A is a HiLog atomic formula.
5. $\mathsf{encode}_a(A \wedge B) = \mathsf{encode}_a(A) \wedge \mathsf{encode}_a(B)$.

For instance, $f(a,X)(b,Y) \wedge X(Y) \wedge Z$ is encoded as:

$$\mathsf{apply}_3(\mathsf{apply}_3(f,a,X),b,Y) \wedge \mathsf{apply}_2(X,Y) \wedge \mathsf{call}(Z)$$

Note that this naive HiLog encoding uses essentially one wrapper predicate per arity. For a Prolog-style implementation, this poses an even greater challenge than F-logic, since all predicate-level indexing is lost. To overcome this problem, two kinds of compiler optimizations can be used: unification factoring [14] and specialization [30]. They both are source-level transformations aimed at improving predicate-level indexing. These techniques are discussed in Section 4.

### 2.3 Transaction Logic

An important aspect of an object-oriented language is the ability to update the internal states of objects. In this respect, F-logic is only partly object-oriented, since it is just a query language. To address this problem, [23] introduced techniques based on preserving the history of object states, so different object states can be distinguished through the extra state argument. However, such techniques do not support modular design. For instance, one cannot define more and more complex update transactions using the previously defined subroutines.

In our view, subroutines are fundamental to programming, and any practical proposal for dealing with updates in a logic-based programming language must address this issue. *Transaction Logic* [8, 7, 9] is one such proposal, which provides a comprehensive theory of updates in logic programming. The utility of Transaction Logic has been demonstrated in various applications ranging from database updates, to robot action planning, to reasoning about actions, to workflow analysis, and many more [8, 10, 12].

In FLORA 2.0, F-logic and Transaction Logic are integrated along the lines of the proposal in [21], and the corresponding implementation issues are described in Section 4. In Transaction Logic, both actions (transactions) and queries are represented as predicates. In the context of F-logic, transactions are expressed as object methods. Underlying Transaction Logic are just a few basic ideas:

1. *Execution* $\equiv$ *Truth.* Execution of an action is tantamount to it being true on a *path*, *i.e.*, a sequence of database states that represent the execution trace.
2. *Elementary Updates.* These are the building blocks for constructing complex transactions. Their behavior can be specified by a separate program (*e.g.*, in the C language) or via a set of axioms. In this paper, we shall use only two types of elementary updates: insert and delete.
3. *Atomicity of Updates.* A transaction should either execute entirely (in which case it is true along the execution path) or not at all. Although common in databases, this behavior is not typical in logic programming, where assert and retract are not backtrackable.

The following program is a FLORA 2.0 adaptation of the block-stacking program from [8]. Here, the action stack is defined as a Boolean method of a robot. The "#"-sign marks transactional methods that change the database state.

```
R[#stack(0,X)]  :- R:robot.
R[#stack(N,X)]  :- R:robot, N > 0,
                   Y[#move(X)], R[#stack(N-1,Y)].
Y[#move(X)]     :- Y:block, Y[clear], X[clear], X[wider(Y)],
                   del(Y[on→Z]), ins(Z[clear]), ins(Y[on→X]), del(X[clear]).
```

Informally, the program says that to stack a pyramid of N blocks on top of block X, the robot must find a block Y, move it onto X, and then stack N-1 blocks on top of Y. To move Y onto X, both of them must be "clear" (*i.e.*, with no block on top), and X must be wider than Y. If these conditions are satisfied,

the database will be updated accordingly (ins and del are elementary insert and delete transactions, respectively).

Note that because of the non-backtrackable nature of Prolog updates, using assert and retract to translate the ins and del transactions in the above program would not work properly. However, backtrackable updates can be implemented efficiently in XSB at the engine level, due to XSB's use of tries — a special data structure for storing dynamic data. Transaction Logic provides semantics to this type of updates.

# 3 Implementation Issues

## 3.1 Transactions in a Tabling Environment

As mentioned in Section 2.1, translation from F-logic to predicate calculus requires tabling all the wrapper predicates used for flattening. It turns out, however, that tabling and database updates are fundamentally at odds: tabling has the effect that whenever the same query is repeated, it is not evaluated and instead the previously computed answers are returned. Even a subsumed query does not necessarily need to be evaluated. Its answers can be computed from the answers for the corresponding subsuming query. Obviously, this hurts the semantics of update transactions and other procedures that have side effects. To see the problem, consider the following program:

```
:- table p/1.        p(X):-write(X).
```

The first time p(a) is called, the system will print out "a" and return the answer yes. However, if p(a) is called the second time, the system will only answer yes without the "side effect" of "a" being printed out.

This problem implies that update transactions in Transaction Logic should *not* be translated using tabled predicates. Moreover, a tabled predicate p should not depend (directly or indirectly) on an update transaction q, since the semantics of such dependency is murky: the first call to p will execute q while subsequent calls might not. Therefore, FLORA must check that regular F-logic methods and attributes do not depend on update transactions. A special syntax is introduced to help FLORA perform proper translation: transactional methods are preceded by a "#"-sign to distinguish them from regular F-logic methods. Primitive update transaction, such as insertion and deletion, also look special:

```
ins(smith:professor[teach(1999,fall)→cse100])
del(cse200[taught_by(1999,spring)→david])
```

A more difficult problem arises when a transaction changes the base facts that a tabled predicate depends on. In this case, the changes should propagate to all answers that are already tabled for this predicate. This is similar to the view maintenance problem in databases, but the overhead associated with database view maintenance methods is unacceptable for fast in-memory logic engines. Currently, FLORA takes a rather drastic approach of abolishing all tables and letting subsequent queries rebuild them. However, this problem is not specific to FLORA, and a more efficient solution can be developed at the XSB engine level.

### 3.2 Problems with Naive Translation of HiLog and F-logic

**Choice Points and Indexing.** In Section 2 we described the naive translation from F-logic and HiLog into classical predicate calculus. Such translation, however, cannot be the basis for practical implementation. The first problem is that the naive translation lays down too many choice points in the top-down execution tree and thus causes excessive backtracking. Consider the following program and its encoding using the apply predicate (we consider translation of HiLog, because it illustrates the problem more dramatically):

```
p(X,Y) :- f(X), g(Y).        apply(p,X,Y) :- apply(f,X), apply(g,Y).
s(X,Y) :- p(X,Y).            apply(s,X,Y) :- apply(p,X,Y).
```
(3)

If apply(p,X,Y) is evaluated, it will unify with all the rules even though its unification with the last rule is bound to fail. In large programs this might cause a serious performance penalty.

Degradation of indexing is another source of performance penalty. Typically, a deductive system indexes on the predicate name plus one of the arguments, *e.g.*, the first. In the naive translation, however, predicate-level indexing is lost, because there are too few predicates used. For instance, in the above example, the translated program has no indexing mechanism corresponding to the first-argument indexing in predicates p and s in the original program.

These problems are not new to logic programming. To tackle them, XSB has developed compiler optimization techniques known as specialization [30] and unification factoring [14], which both perform source-to-source transformation.

Specialization takes place when a goal can only unify with a subset of the candidate rules. By replacing this goal's predicate with a different predicate that can only unify with the heads of *some* of the rules, specialization throws out the unnecessary choice points. For instance, performing specialization on the translated program in (3) yields the following more efficient program, where some occurrences of the predicate apply are replaced with apply_1:

```
apply(p,X,Y) :- apply(f,X), apply(g,Y).      apply(s,X,Y) :- apply_1(X,Y).
apply_1(a,X) :- apply(f,X), apply(g,Y).
```

In contrast to specialization, unification factoring is driven by the patterns in rule heads. The idea is to factor out common function symbols to save on unification and achieve better indexing. Consider the following program:

```
p(apply(a),X) :- q(X).      p(apply(b),X) :- r(X).
```

and the query ?- p(apply(X),Y). Here unification for apply has to take place once with each rule head. However, this repeated unification can be avoided if the same goal is executed against the following transformed program:

```
p_apply(a,X) :- q(X).       p(apply(X),Y) :- p_apply(X,Y).
p_apply(b,X) :- r(X).
```

Because apply is used to encode HiLog terms, common functors, as in the above example, occur very frequently in a translated FLORA program. It turns out that the native XSB unification factoring performs quite well with FLORA-translated programs. XSB specialization, however, exhibits subtle problems.

**Double Tabling.** The first problem with specialization is tabling. In HiLog translation, it is not very clear how a tabling directive like :- table p/2 should be translated. If FLORA handles this by tabling apply/3, then XSB specialization may cause "double tabling" — a situation where certain predicates are tabled unnecessarily. For instance, consider the following program (which computes transitive closure) and its naive encoding:

```
:- table p/2.                  :- table apply/3.
p(a,b).                        apply(p,a,b).
p(b,c).                        apply(p,b,c).                          (4)
t(X,Y) :- p(X,Y).              apply(t,X,Y) :- apply(p,X,Y).
t(X,Y) :- p(X,Z), t(Z,Y).      apply(t,X,Y) :- apply(p,X,Z), apply(t,Z,Y).
```

XSB specialization on the translated program (4) would yield the following:

```
:- table apply/3.
:- table apply_1/2.        :- table apply_2/2.
apply_1(a,b).              apply_2(X,Y) :- apply_1(X,Y).
apply_1(b,c).              apply_2(X,Y) :- apply_1(X,Z), apply_2(Z,Y).
apply(p,a,b).              apply(t,X,Y) :- apply_1(X,Y).
apply(p,b,c).              apply(t,X,Y) :- apply_1(X,Z), apply_2(Z,Y).
```

Being essentially another copy of apply(t,X,Y), tabling the tuples of apply_2(X,Y) is redundant, although this caching is needed to guarantee termination of the specialized program. The size of the compiled code is also considerably larger than the original.

**Meta-Programming.** Yet another problem is due to meta-programming, which tends to produce programs that preclude XSB specialization. To see the crippling effect of meta-rules on XSB specialization, consider the following program and its naive translation:

```
p(a).                      apply(p,a).
p(b).                      apply(p,b).
X(Y) :- X=p, Y=c.          apply(X,Y) :- X=p, Y=c.                    (5)
t(X) :- p(X).              apply(t,X) :- apply(p,X).
```

XSB specialization on the previous translated program (5) looks as follows:

```
apply(p,a).                apply_1(p,a).
apply(p,b).                apply_1(p,b).
apply(X,Y) :- X=p, Y=c.    apply_1(X,Y) :- X=p, Y=c.
apply(t,X) :- apply_1(p,X).
```

In this program, the predicate apply_1(p,X) still has to unify with all the apply_1 facts and rules. Not only the unification on p is repeated, but indexing on the first argument in the original program is lost as well.

Note that although so far we have been illustrating the XSB specialization problems using HiLog only, F-logic exhibits the same problem. Consider the following F-logic program and its naive translation:

```
obja[atta→vala].                  fd(atta,obja,[ ],vala).
objb[atta→valb].                  fd(atta,objb,[ ],valb).
objc[X→Y] :- X=atta, Y=valc.      fd(X,objc,[ ],Y) :- X=atta, Y=valc.
O[attb↠{X}] :- O[atta→X].         mvd(attb,O,[ ],X) :- fd(atta,O,[ ],X).
```
(6)

It is easy to see that the translation is just another version of the previous HiLog program (5) and thus it cripples XSB specialization just as badly.

The next section proposes a new kind of specialization, called *skeleton-based specialization*, which is used in FLORA 2.0 to optimize source-level translation for F-logic and HiLog. The system is designed in such a way that skeleton-based specialization and XSB specialization compliment each other.

# 4 Solutions

As explained in Section 3, a major problem with the naive translation of F-logic and HiLog is the loss of indexing and while XSB unification factoring performs well for the translated programs, specialization often fails to yield any improvements and, in some cases, it might even cause unnecessary overhead. In this section we propose *skeleton-based specialization*, which supplements the native XSB specialization and fixes the aforesaid problems.

## 4.1 Skeleton-Based Specialization Algorithm

**Definition 1 (Skeleton).** *Given a HiLog term* T*, its* skeleton Skel(T) *is an abstract view of the syntactic structure of* T. Skel(T) *is defined as follows:*

1. Skel(T) = T, *if* T *is a constant.*
2. Skel(T) = _, *if* T *is a variable.*
3. Skel(T) = Skel(F)/n, *if* T = $F(T_1,...,T_n)$.

*Example 1 (Skeletons of HiLog Terms).*

1. Skel(f) = f
2. Skel(X(a,b)(Y)) = _/2/1
3. Skel(X(f(Y))) = _/1

The algorithm in Figure 1 describes FLORA skeleton-based specialization. It applies to F-logic and HiLog translation separately, since the set of wrapper predicates used for F-logic translation is disjoint from those wrapper predicates used for HiLog predicates.

First we explain the algorithm in the context of HiLog translation. It takes a FLORA program as input and yields an equivalent program in predicate logic; the algorithm has the following steps:

```
Input: a FLORA program F consisting of rules (including facts)
Output: an XSB program that encodes F
1    HL := {L | L is a literal in a rule head of F};
2    BL := {L | L is a literal in a rule body of F};
3    HS := {Skel(L) | L ∈ HL};
4    BS := {Skel(L) | L ∈ BL};
5    for each skeleton S ∈ HS ∪ BS do seq(S) := a unique integer;
6    for each rule H :- B from the input program F do {
7        H' := flatten(H,Skel(H));
8        B' := B;
9        for each literal L ∈ B' do L := flatten(L,Skel(L));
10       output the rule H' :- B';
11   }
12   for each literal H ∈ HL do {
13       H' := naive(H);
14       H'' := flatten(H,Skel(H));
15       output the rule H' :- H'';
16   }
17   for each literal L ∈ BL do
18       for each rule H :- B from the input program F do
19           if L unifies with H with the mgu θ and Skel(L) ≠ Skel(H) then {
20               H' := flatten(Hθ,Skel(L));
21               B' := B;
22               for each literal T ∈ B' do {
23                   S := Tθ;
24                   if Skel(S) ∈ BS
25                       then T := flatten(S,Skel(S));
26                       else T := flatten(S,Skel(T));
27               output the rule H' :- B';
28           }
```

**Fig. 1.** Skeleton-Based Specialization Algorithm

**Skeleton Analysis** (Lines 1 – 5). First we collect all the literals in rule heads into the set HL and all the literals in rule bodies into the set BL.[2] Then, the algorithm computes the set of skeletons HS and BS for each literal in HL and BL, respectively. Each unique skeleton in the union of HS and BS is assigned a unique sequence number.

The rest of the algorithm consists of three main tasks: *flattening*, *trap rule generation*, and *instantiation*.

**Flattening** (Lines 6 – 11). The purpose of flattening is to eliminate unnecessary wrapper predicates and unification. Let $S = X/n_1/\ldots/n_k$, where X is either

[2] Each HiLog literal is assumed to have the functor part and the arity. Propositional constants are treated as 0-ary literals, *e.g.*, p().

"_" or a constant, and L be of the form $T(T_{1n_1},\ldots,T_{n_1n_1})\ldots(T_{1n_k},\ldots,T_{n_kn_k})$. The transformation procedure flatten(L,S) then does the following: Let n be the sequence number assigned to the skeleton S, then the wrapper predicate used to encode the HiLog literal L is apply_n, which is unique across HiLog translation. Next, if X is a constant in $X/n_1/\ldots/n_k$, then so must be T (in Lines 7, 14 and 25 the skeleton argument of flatten is that of the literal argument whereas in Lines 20 and 26 the skeleton either subsumes or is the same as that of the literal) and flatten(L,S) yields $\mathsf{apply\_n}(E_{1n_1},\ldots,E_{n_1n_1},\ldots,E_{1n_k},\ldots,E_{n_kn_k})$. Otherwise, X is "_" and T might be any HiLog term, then flatten(L,S) will return $\mathsf{apply\_n}(E,E_{1n_1},\ldots,E_{n_1n_1},\ldots,E_{1n_k},\ldots,E_{n_kn_k})$, where E, $E_{ij} = \mathsf{encode_t}(T)$, $\mathsf{encode_t}(T_{ij})$, respectively, $\mathsf{encode_t}$ is the naive encoding of HiLog terms described in Section 2.2. For instance, if the sequence number assigned to the skeleton f/1/2 is 2, then flatten(f(Y)(a,Z),f/1/2) will produce apply_2(Y,a,Z). The reason why the functor symbol f can be omitted is because it is already encoded in the sequence number for the skeleton.

**Trap Rule Generation** (Lines 12 – 16). These steps generate rules to "trap" the naive encoding of literals. The translation outputs a rule whose head is the naive encoding of the original rule-head, while the body is the result of flattening the head. For instance, the trap rule for $\mathsf{f(Y)(a,Z)} :- \mathsf{body}$ is like $\mathsf{apply(apply(f,Y),a,Z)} :- \mathsf{apply\_2(Y,a,Z)}$. Trap rule generation is indispensable for inter-module communications in FLORA. Since specialization in principle has no knowledge of other modules, calls referring to other modules have to be encoded using the naive translation. Due to space limits, we will not elaborate on this topic further.

**Instantiation** (Lines 17 – 28). Even when two literals unify, their encodings might not unify after flattening. For instance, X(Y) and f(a)(Z) unify, but their flattened forms, *e.g.*, apply_1(X,Y) and apply_2(a,Z) (with respect to the skeletons _/1 and f/1/1, respectively), do not unify.

Instantiation ensures that unifiability is preserved after specialization. The idea is that if a body literal unifies with the head of a rule, R, using the mgu $\theta$, but the two literals have different skeletons, then a new rule, $R\theta$, must be generated. For instance, consider the following program:

$$\mathsf{g(X)} :- \mathsf{p(X)}. \qquad \mathsf{Y(Z)} :- \mathsf{q(Y,Z)}.$$

Here p(X) will be flattened as apply_1(X) and Y(Z) as apply_2(Y,Z). Because p(X) unifies with $\mathsf{Y(Z)} :- \mathsf{q(Y,Z)}$, this rule must be instantiated using the substitution Y/p, yielding $\mathsf{p(Z)} :- \mathsf{q(p,Z)}$. Specializing this rule yields $\mathsf{apply\_1(Z)} :- \mathsf{apply\_2(p,Z)}$, which ensures that the semantics of the original program is preserved.

However, rule instantiation might generate body literals with *new* skeletons that have not been seen before in the original program. Thus, instantiation might have to be applied again, using these new body literals. This opens up the possibility of an infinite instantiation process. For instance, in the following program:

```
g(X) :- p(X).        Y(Z) :- Y(Z)(Z).
```

when the second rule is instantiated with Y/p (the mgu of p(X) and Y(Z)), a new rule p(Z) :- p(Z)(Z) is generated. The literal p(Z)(Z) has a completely new skeleton: p/1/1. If p(X)(X) is flattened with respect to p/1/1, the rule Y(Z) :- Y(Z)(Z) has to be instantiated with `Y/p(X)`, the mgu of p(X)(X) and Y(Z). Thus yet another new skeleton p/1/1/1 will emerge, and so on.

Lines 24 – 26 in the algorithm are designed to ensure termination of the instantiation process. The solution is simple: the quality of specialization is traded in for termination. When a literal with a new skeleton shows up in a newly instantiated rule, its skeleton must extend the skeleton of that literal before instantiation. Thus, we can flatten the instantiated literal with respect to the skeleton of the original literal. Unifiability is also preserved by such translation. For instance, specializing the above example yields the following program (where the trap rules are omitted):

```
apply_1(X) :- apply_2(X).          apply_2(X) :- apply_4(p,X,X).
apply_3(Y,Z) :- apply_4(Y,Z,Z).    apply_4(Y,Z,Z) :- apply_4(apply(Y,Z),Z,Z).
```

### 4.2 Putting it All Together

For the translated program (4), which computes transitive closure, the result of skeleton-based specialization is as follows:

```
:- table apply_2/2.
apply_1(a,b).          apply_2(X,Y) :- apply_1(X,Y).
apply_1(b,c).          apply_2(X,Y) :- apply_1(X,Z), apply_2(Z,Y).
```

The following program is the result of skeleton-based specialization of the program shown in (5):

```
apply_1(a).                        apply_3(X) :- apply_1(X).
apply_1(b).                        apply_1(X) :- p=p, X=c.
apply_2(X,Y) :- X=p, Y=c.
```

Note that although we illustrate the idea of skeleton-based specialization using HiLog translation, our algorithm applies to F-logic translation as well. In fact, the translation views F-logic literals as just another kind of HiLog literals, which just happen to use different wrapper predicates.

For instance, a slight variation of the naive F-logic translation can convert O[M→V] into the HiLog literal M(O,V) and then further convert it to predicate logic using the wrapper predicate fd instead of apply. Likewise, O[M↠V] can be converted to M(O,V) and then to predicate calculus using mvd as a wrapper. Therefore, skeleton-based specialization can be performed on HiLog and F-logic independently. The only part of the algorithm that needs to be changed is the prefix used to construct the wrappers. For instance, instead of apply_2 we would use fd_2. Thus, the result of applying skeleton-based specialization to the program (6) would be the following (where the trap rules are omitted):

```
fd_1(obja,vala).                    mvd_1(O,X) :- fd_1(O,X).
fd_1(objb,valb).                    fd_1(objc,Y) :- atta=atta, Y=valc.
fd_2(X,objc,Y) :- X=atta, Y=valc.
```

Our experiments show that even for small programs discussed in this section FLORA skeleton-based specialization can speed up programs by a factor of 2.1, whereas XSB native specialization reduces execution time only by a factor of 1.85. A more detailed comparison will be reported in the full version of this paper. Nevertheless, as said earlier, FLORA specialization is not intended to replace XSB specialization. Instead, it is used as a first-line optimization technique. Then the FLORA-translated program is further optimized through the native XSB specialization and unification factoring.

Another observation about FLORA specialization is that better-quality specialization is possible with more detailed skeleton representation. Indeed, considering HiLog terms as trees, we could define skeletons as the abstract view of their structures at some depth level. For example, a two-level skeleton for f(X)(X,a,f(b)) would be f/(_)/(_,a,(f/1)). There is a subtle relationship, though, between the amount of detail preserved in skeletons and the quality of specialized programs. More detailed skeletons normally mean better specialized programs and thus better performance, but longer compilation time and larger program size.

## 5 Conclusion

This paper discusses techniques for building efficient DOOD systems by translation into lower-level Prolog syntax and utilizing an existing tabling logic engine, such as XSB [29]. The feasibility of our approach has been demonstrated by the F-logic based FLORA system, which delivers very encouraging performance. (Performance results will be included in the full version of this paper.) We also discuss the compiler optimization techniques that were used to achieve this performance; some of them are just native XSB optimizations, while others are designed specifically for FLORA. Due to lack of space we omitted a number of other implementation issues, such as the FLORA module system and performance optimizations related to handling path expressions. Details can be found at `http://www.cs.sunysb.edu/~guizyang/papers/floratech.ps`

**Acknowledgement** We would like to thank Hasan Davulcu, Kostis Sagonas, C.R. Ramakrishnan, and David S. Warren for their patience in explaining us the intricacies of XSB optimization techniques. We are also grateful to Bertram Ludäscher and the anonymous referees for the very helpful comments.

## References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kaufmann, San Francisco, CA, 2000.

2. M.L. Barja, A.A.A. Fernandes, N.W. Paton, A.H. Williams, A. Dinn, and A.I. Abdelmoty. Design and implementation of ROCK & ROLL: A deductive object-oriented database system. *Information Systems*, 20(3):185–211, 1995.
3. C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, April 1991.
4. T. Berners-Lee. Semantic web road map. http://www.w3.org/DesignIssues/Semantic.html, September 1998.
5. T. Berners-Lee. The semantic toolbox: Building semantics on top of XML-RDF. http://www.w3.org/DesignIssues/Toolbox.html, May 1999.
6. A.J. Bonner and M. Kifer. Transaction logic programming. In *Int'l Conference on Logic Programming*, pages 257–282, Budapest, Hungary, June 1993. MIT Press.
7. A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.
8. A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.
9. A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.
10. A.J. Bonner and M. Kifer. Results on reasoning about updates in transaction logic. In [16]. 1998.
11. W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.
12. H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, pages 25–33, Seattle, Washington, June 1998.
13. H. Davulcu, G. Yang, M. Kifer, and I.V. Ramakrishnan. Design and implementation of the physical layer in webbases: The XRover experience. In *Int'l Conference on Computational Logic (DOOD-2000 Stream)*, July 2000.
14. S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D.S. Warren. Unification factoring for efficient evaluation of logic programs. In *ACM Symposium on Principles of Programming Languages*, 1995.
15. S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for RDF. In *QL'98 - The Query Languages Workshop*, December 1998.
16. B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors. *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, 1998.
17. J. Frohn, R. Himmeroeder, P.-Th. Kandzia, G. Lausen, and C. Schlepphorst. FLORID – A prototype for F-logic. In *Proc. Intl. Conference on Data Engineering (ICDE, Exhibition Program)*. IEEE Computer Science Press, 1997.
18. R.V. Guha, O. Lassila, E. Miler, and D. Brickley. Enabling inferencing. In *QL'98 - The Query Languages Workshop*, December 1998.
19. A. Gupta, B. Ludäscher, and M. E. Martone. Knowledge-based integration of neuroscience data sources. In *12th Intl. Conference on Scientific and Statistical Database Management (SSDBM)*, Berlin, Germany, July 2000. IEEE Computer Society.
20. M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, and Stefan Eherer. ConceptBase – A deductive object base for meta data management. *Journal of Intelligent Information Systems*, February 1995.

21. M. Kifer. Deductive and object-oriented data languages: A quest for integration. In *Int'l Conference on Deductive and Object-Oriented Databases*, volume 1013 of *Lecture Notes in Computer Science*, pages 187–212, Singapore, December 1995. Springer-Verlag. Keynote address at the 3d Int'l Conference on Deductive and Object-Oriented databases.
22. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.
23. G. Lausen and B. Ludäscher. Updates by reasoning about states. In *2-nd International East/West Database Workshop*, Klagenfurt, Austria, September 1994.
24. M. Liu. A deductive object base language. *Information Systems*, 21(5):431–457, 1996.
25. B. Ludäscher. Tour de FLIP. The FLIP manual, 1998.
26. B. Ludäscher, A. Gupta, and M. E. Martone. A mediator system for model-based information integration. In *Int'l Conference on Very Large Data Bases*, Cairo, Egypt, 2000. system demonstration.
27. B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing semistructured data with FLORID: A deductive object-oriented perspective. *Information Systems*, 23(8):589–613, 1998.
28. Mozilla RDF/Enabling inference. http://www.mozilla.org/rdf/doc/inference.html, 1999.
29. K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD Conference on Management of Data*, pages 442–453, New York, May 1994. ACM.
30. K. Sagonas and D.S. Warren. Efficient execution of HiLog in WAM-based Prolog implementations. In *Int'l Conference on Logic Programming*, 1995.
31. T. Swift and D. S. Warren. An abstract machine for SLG resolution: Definite programs. In *Int'l Logic Programming Symposium*, Cambridge, MA, November 1994. MIT Press.
32. H. Tamaki and T. Sato. OLD resolution with tabulation. In *Int'l Conference on Logic Programming*, pages 84–98, Cambridge, MA, 1986. MIT Press.
33. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.
34. K. Yokota and H. Yasukawa. Towards an integrated knowledge-base management system. In *Proceedings of the Int'l Conference on Fifth Generation Computer Systems*, pages 89–109, June 1992.