$\mathcal{F}$LORA: The Secret of Object-Oriented Logic Programming

Bertram Ludäscher   Guizhen Yang   Michael Kifer

July 13, 2001

Contents

1 Introduction

$\mathcal{F}$LORA is a sophisticated F-logic to XSB compiler. It translates a program written in the F-logic language [3] (which must be in a file with extension `.flr`, *e.g.*, `file.flr`) and outputs a file with extension `.P` (*e.g.*, `file.P`), which is a regular XSB program. This program is then passed to XSB for compilation (yielding `file.O`) and execution.

The current version of $\mathcal{F}$LORA was implemented by Guizhen Yang, but its origins trace back to the FLIP compiler developed by Bertram Ludäescher, and the basic architectures of the two compilers are similar. However, unlike FLIP, $\mathcal{F}$LORA is a complete application development platform with many features not found in FLIP. It has a much more optimized compiler, and its tokenizer and parser are very different from FLIP's.

The programming language supported by $\mathcal{F}$LORA is a dialect of F-logic that is mostly compatible with the extensions introduced in FLORID, a C++-based F-logic system developed at Freiburg University.[1] In particular, $\mathcal{F}$LORA fully supports the versatile syntax of FLORID path expressions. However, $\mathcal{F}$LORA has numerous extensions of its own, and some features differ significantly.

$\mathcal{F}$LORA is part of the official distribution of XSB beginning with version 2.0. It is organized as an XSB package and lives in the directory

```
<xsb-installation-directory>/packages/flora/
```

$\mathcal{F}$LORA is fully integrated into the XSB system, including its module system. In particular, $\mathcal{F}$LORA modules can invoke predicates defined in other XSB modules, and regular XSB modules can query the objects defined in $\mathcal{F}$LORA modules. At present, XSB is the only platform where $\mathcal{F}$LORA can run, because it heavily relies on tabling and the well-founded semantics for negation that at the moment are available only in XSB.

As mentioned earlier, an XSB programmer can invoke $\mathcal{F}$LORA objects from other XSB programs. However, the easiest way to get a feel of the system is to start $\mathcal{F}$LORA shell and begin to enter queries interactively. To this end, you must first invoke XSB and then load the `flora` package:

```
foo>  xsb
...  XSB loading messages omitted ...
| ?- [flora].
[flora loaded]
| ?-
```

At this point, it is possible to use a limited number of $\mathcal{F}$LORA commands, but to run queries you must enter the $\mathcal{F}$LORA command loop:

```
| ?- flora_shell.
...  FLORA messages omitted ...
flora ?-
```

[1] See `http://www.informatik.uni-freiburg.de/~dbis/florid/` for more details.

At this point, *F*LORA takes over and F-logic syntax becomes the norm. To get back to the XSB command loop, type `Control-D` or

```
| ?- end.
```

*F*LORA comes with a number of demo programs that live in

```
<xsb-installation-directory>/packages/flora/demos/
```

The demos can be run by issuing the command "`rundemo(demo-filename).`" at the *F*LORA prompt, *e.g.*,

```
rundemo(flogic_basics).
```

There is no need to change to the demo directory.

2 *F*LORA Shell Commands

The following *F*LORA shell commands are supported:

```
help                      :  show this info
compile('FILE')           :  compile FILE.P; create FILE.O
flcompile('FILE')         :  compile FILE.flr; create FILE.P and FILE.O
flcompile('FILE',[...])[2] :  flcompile('FILE') with options [...]
flconsult('FILE')         :  compile FILE.flr, then consult FILE.P
flconsult('FILE',[...])   :  flconsult('FILE') with options [...]
flload('FILE[.EXT]')[3]    :  consult FILE.flr, FILE.P or FILE.O
['FILE[.EXT]',...]        :  consult a list of .flr, .P, or .O files
dyncompile('FILE')        :  compile FILE.flr to dynamic code
dyncompile('FILE',[...])  :  dyncompile('FILE') with options [...]
dynconsult('FILE')        :  dyncompile FILE.flr, then dynamically load FILE.P
dynconsult('FILE',[...])  :  dynconsult('FILE') with options [...]
dynload('FILE[.EXT]')[4]   :  dynamically load FILE.flr or FILE.P
<'FILE[.EXT]',...>        :  dynload a list of .flr or .P files
rundemo('FILE')           :  flconsult a demo from FLORA demos directory
rundemo('FILE',[...])     :  rundemo('FILE') with options [...]
abolish_all_tables[5]      :  flush all tabled data
all                       :  show all solutions at once (default)
one                       :  show solutions one by one
maxerr(all/N)             :  set/show the max number of errors FLORA reports
end                       :  say Ciao to FLORA
halt                      :  quit FLORA and XSB
```

All commands with a FILE argument passed to them use the XSB `library_directory` predicate to search for the module, except that the command `rundemo(FILE)` first looks for `FILE` in the $\mathcal{F}$LORA demo directory. In general, all XSB commands can be executed from $\mathcal{F}$LORA shell, if the corresponding XSB library has already been loaded.

After a syntax error, parsing error, or compiling error, $\mathcal{F}$LORA shell will discard tokens read from the current input stream until the end of file or a rule delimiter (.) is encountered. If $\mathcal{F}$LORA shell seems to hang forever after the prompt:

```
[FLORA: discarding tokens]
```

hitting the Enter key once, then entering a "." character and Enter again will normally reset the current input buffer and cause $\mathcal{F}$LORA issue a command prompt:

```
flora ?-
```

3 F-logic and $\mathcal{F}$LORA by Example

In the future, this section will contain a number of small introductory examples illustrating the use of F-logic and $\mathcal{F}$LORA. Meanwhile, the reader is referred to the excellent tutorial written by the members of the FLORID project.[6] Since $\mathcal{F}$LORA and FLORID share much of the same syntax, most examples in that tutorial are also valid $\mathcal{F}$LORA programs.

4 Inside $\mathcal{F}$LORA

$\mathcal{F}$LORA consists of the following modules:

- `flrshell.P`: top-level module that provides the $\mathcal{F}$LORA shell commands for compiling and consulting $\mathcal{F}$LORA programs (`flcompile/1`, `flconsult/1`), for setting the output mode (`all/0` or `one/0` solution(s) at a time), and – last but not the least – for directly issuing queries against the loaded database/program (see Section 2 for a full description of shell commands).
- `flrtokens.P`: $\mathcal{F}$LORA tokenizer.
- `flrparser.P`: DCG parser for F-logic.
- `flrcompiler.P`: $\mathcal{F}$LORA compiler that translates F-logic to XSB.
- `flrutils.P`: miscellaneous utility predicates.

[2]Currently supported is equality checking option: eqlevel(N), N=0,1.
[3]File extension is optional, but must be .flr, .P or .O if supplied.
[4]File extension is optional, but must be .flr or .P if supplied.
[5]Tables need to be flushed if the database has been changed since last evaluation.
[6]See `http://www.informatik.uni-freiburg.de/~dbis/florid/` for more details.

Additional libraries are located in the `lib/` subdirectory, and there is also a number of files in the `closure/` subdirectory that serve as headers and trailers that are automatically attached to the `*.P` files by $\mathcal{F}$LORA compiler (explained later).

4.1 How $\mathcal{F}$LORA Works

Overview. As an F-logic-to-XSB compiler, $\mathcal{F}$LORA first parses its argument file and then compiles it to XSB syntax. For instance the command

```
flora ?- flconsult(myprog).
```

compiles the program `'myprog.flr'` into the XSB file `'myprog.P'`. Take a look at this file to see what has become of your F-logic program! The compilation consists mainly of a flattening procedure sketched below. Next, `'myprog.P'` is compiled by XSB, yielding byte-code `'myprog.O'`, which is then loaded and executed. If `'myprog.flr'` contains queries, they are immediately executed by XSB (provided there are no errors).

The main purpose of the $\mathcal{F}$LORA shell, however, is to allow the evaluation of ad-hoc F-logic queries. For example, after having requested the execution of the `'default.flr'` file from the demo directory (using the command `flora ?- rundemo(default).`), you may ask

```
flora ?-  X..kids[                    % Whose kids
                self -> K;             % ... (list them by name)
                hobbies ->>            % ... have hobbies
                {H:dangerous_hobby}    % ... that are dangerous?
          ].
```

$\mathcal{F}$LORA will parse, flatten, and evaluate this query in the same way as the queries in a source file.

Flattening F-logic. Consider, e.g., the following complex F-logic molecule, representing facts about the object `mary` (the syntax of F-logic is given in Section 5.1):

```
mary:employee[age->29;kids->>{tim,leo};salary@(1998)->a_lot].
```

As described in [3], any complex F-logic molecule can be decomposed into a conjunction of simpler F-logic atomic formulas. These latter atoms can be directly represented using Prolog syntax. For the different kinds of F-logic atoms we use different Prolog predicates. For instance, the result of translating the above F-molecule might be:

```
'_$_$_flora_isa'(mary,employee).                % mary:employee.
'_$_$_flora_fd'(mary,@(age),29).                % mary[age->29].
'_$_$_flora_mvd'(mary,@(kids),tim).             % mary[kids->>{tim}].
'_$_$_flora_mvd'(mary,@(kids),leo).             % mary[kids->>{leo}].
'_$_$_flora_fd'(mary,@(salary,1998),a_lot).     % mary[salary@(1998)->a_lot].
```

Closure Axioms. The flattening process alone is not enough to convert an F-logic program into Prolog, because of the semantics "hidden" behind the notions of the subclass relationship, inheritance, and scalar methods. This semantics is captured through the facts and rules called *closure axioms*, which must be explicitly added to the flattened user program. Closure axioms are static and reside in the subdirectory `closure/`; these files are appended to every `*.P` file by the ℱLORA compiler. These closure rules also perform the following tasks:

- Transitive closure of "`::`" (the subclass relationship). A runtime check warns about cycles in the subclass hierarchy.
- Closure of "`:`" with respect to "`::`", i.e., if $X : C, C :: D$ then $X : D$.
- Perform monotone and non-monotone inheritance.
- Make sure that scalar methods are, indeed, scalar.

4.2 ℱLORA vs. FLORID

The syntax of ℱLORA and some of its design decisions are borrowed from FLORID, an F-logic interpreter developed at Freiburg University, Germany. For more information on Florid please visit the project home page at: `http://www.informatik.uni-freiburg.de/~dbis/florid/`. The following is a list of differences between these two systems.

- FLORID
  - (Semi-)naive bottom-up evaluation.
  - "Hard-wired" closure axioms.
  - Nonmonotonic inheritance (trigger semantics).
  - C++ based system.
- ℱLORA
  - Translation of F-logic into XSB rules.
  - Top-down evaluation of the generated rules. When tabling is used, the compiled programs can be much more efficient than the corresponding FLORID programs.
  - Closure axioms implemented as Prolog rules and are easy to experiment with.
  - Non-monotonic inheritance implemented using closure axioms and the well-founded semantics.
  - Flora has a module system that fully integrates with the XSB module system.
  - Flora programs have full access to the underlying XSB system, and vice-versa.

5 Syntax of ℱLORA

The following is adopted from [4].

5.1 Basic F-logic Syntax

- *Symbols*: The F-logic alphabet of *object constructors* consists of the sets $\mathcal{F}$(function symbols), $\mathcal{P}$(predicate symbols including $\doteq$), and $\mathcal{V}$(variables). Variables are denoted by capitalized symbols or an underscore followed by zero or more letters and/or digits (e.g., X, *Name*, _, _$v5$).[7] All other symbols, including the constants (which are 0-ary object constructors), are symbols that start with a lowercase letter (e.g., $a, john$). Constants can also start with uppercase and include non-alphanumeric symbols, but then they must be enclosed in single quotes (*e.g.*, `'AB@*c'`).

  In addition to the usual first-order connectives and symbols, there is a number of special symbols:], [, }, {, $\rightarrow$, $\twoheadrightarrow$, $\Rightarrow$, $\Rrightarrow$, **:**, **::**. Later we shall introduce additional symbols used by the inheritance mechanism.

- *Id-Terms/Oids*: [8]

  First-order terms over $\mathcal{F}$ and $\mathcal{V}$ are called *id-terms*, and are used to name objects, methods, and classes. Ground id-terms (*i.e.*, terms with no variables) correspond to *logical object identifiers* (*oids*), also called object *names.*

- *Atomic formulas*: Let O, M, R_i, X_i, C, D, T be id-terms. In addition to the usual first-order atoms, like $p(X_1, \ldots, X_n)$, there are the following basic types of formulas:

  $$(1)\ \ O[M{\rightarrow}R_0] \qquad (2)\ \ O[M{\twoheadrightarrow}\{R_1,\ldots,R_n\}] \qquad (3)\ \ C[M{\Rightarrow}T] \qquad (4)\ \ C[M{\Rrightarrow}T].$$

  (1) and (2) are *data atoms*, which specify that a *method* M applied to an object O yields the result-object R_i. In (1), M is a *single-valued* (or *scalar*) method, i.e., there is at most one R_0 such that $O[M{\rightarrow}R_0]$ holds. In contrast, in (2), M is *multi-valued*, so there can be several result-objects R_i. For $n = 1$ the curly braces can be omitted.

  (3) and (4) denote *signature atoms.* They specify that method M, applied to objects of *class* C, yields results of type T. In (3), M is declared as single-valued, and in (4) as set-valued.

  Objects are classified into classes using *isa-atoms*:

  $$(5)\ \ O\,{:}\,C \qquad\qquad (6)\ \ C\,{::}\,D.$$

  (5) defines that O is an *instance* of class C, while (6) specifies that C is a *subclass* of D.

- *Parameters*: Methods can have arguments, *i.e.*, $M@(P_1, \ldots, P_k)$ is allowed in (1) – (4), where $P_1, \ldots, P_k$ are id-terms, e.g., john[salary@(1998)$\rightarrow$50000].

- *Programs*: F-logic *literals*, *rules*, and *programs* are defined as usual, based on F-logic atoms.

[7]The symbol "_" denotes an anonymous variable, as in Prolog.

[8]Numbers (including integers and floats) may also be used as id-terms. But such use might be confusing and is not recommended.

F-molecules provide a shortcut for specifying properties of the same object. For instance, instead of john:person $\wedge$ john[age$\rightarrow$31] $\wedge$ john[children$\twoheadrightarrow${bob,mary}], we can simply write john : person[age$\rightarrow$31; children$\twoheadrightarrow${bob,mary}].

Example 5.1 (Publications Database) Figure 1 depicts an F-logic representation of a fragment of an object-oriented publications database.

Schema:
conf_p :: paper.
journal_p :: paper.
paper[authors$\Rightarrow\!\!\!\!\!\Rightarrow$person; title$\Rightarrow$string].
journal_p[in_vol$\Rightarrow$volume].
conf_p[at_conf$\Rightarrow$conf_proc].
journal_vol[of $\Rightarrow$journal; volume$\Rightarrow$integer; number$\Rightarrow$integer; year$\Rightarrow$integer].
journal[name$\Rightarrow$string; publisher$\Rightarrow$string; editors@(integer)$\Rightarrow\!\!\!\!\!\Rightarrow$person].
conf_proc[of_conf$\Rightarrow$conf_series; year$\Rightarrow$integer; editors@(integer)$\Rightarrow\!\!\!\!\!\Rightarrow$person].
conf_series[name$\Rightarrow$string].
publisher[name$\Rightarrow$string].
person[name$\Rightarrow$string; affil@(integer)$\Rightarrow$institution].
institution[name$\Rightarrow$string; address$\Rightarrow$string].

Objects:
o_{j1} : journal_p[title$\rightarrow$"Records, Relations, Sets, Entities, and Things"; authors$\twoheadrightarrow$$\{o_{mes}\}$; in_vol$\rightarrow$$o_{i11}$].
o_{di} : conf_p[title$\rightarrow$"DIAM II and Levels of Abstraction"; authors$\twoheadrightarrow$$\{o_{mes}, o_{eba}\}$; at_conf$\rightarrow$$o_{v76}$].
o_{i11} : journal_vol[of$\rightarrow$$o_{is}$; number$\rightarrow$1; volume$\rightarrow$1; year$\rightarrow$1975].
o_{is} : journal[name$\rightarrow$"Information Systems"; editors@(...)$\twoheadrightarrow$$\{o_{mj}\}$].
o_{v76} : conf_proc[of$\rightarrow$vldb; year$\rightarrow$1976; editors$\twoheadrightarrow$$\{o_{pcl}, o_{ejn}\}$].
o_{vldb} : conf_series[name$\rightarrow$"Very Large Databases"].
o_{mes} : person[name$\rightarrow$"Michael E. Senko"].
o_{mj} : person[name$\rightarrow$"Matthias Jarke"; affil@(...)$\rightarrow$$o_{rwt}$].
o_{rwt} : institution[name$\rightarrow$"RWTH_Aachen"].

Figure 1: A Publications Object Base and its Schema Represented Using F-logic

5.2 Path Expressions in the Rule Body

In addition to the basic F-logic syntax, the $\mathcal{F}$LORA system also supports *path expressions* to simplify object navigation along single-valued and multi-valued method applications, and to avoid explicit join conditions [1]. The basic idea is to allow the following *path expressions* wherever id-terms are allowed:

$$(7)\ \ O.M \qquad\qquad (8)\ \ O..M$$

The path expression in (7) is *single-valued*; it refers to the unique object R_0 for which $O[M\rightarrow R_0]$ holds; (8) is a *multi-valued* path expression; it refers to each R_i for which $O[M\twoheadrightarrow\{R_i\}]$ holds. The symbols O and M stand for an id-term or path a expression. Moreover, M can be a method that takes arguments, i.e., $O..M@(P_1, \ldots, P_k)$ is a valid path expression.

In order to obtain a unique syntax and to specify different orders of method applications, parentheses can be used. By default, path expressions associate to the left, so $a.b.c$ is equivalent to $(a.b).c$ and specifies the unique object o such that $a[b \rightarrow x] \wedge x[c \rightarrow o]$ holds (note that $x = a.b$). In contrast, $a.(b.c)$ is the object o' such that $b[c \rightarrow x'] \wedge a[x' \rightarrow o']$ holds (here, $x' = b.c$). In general, these can be different objects. Note that in $(a.b).c$, b is a method name, whereas in $a.(b.c)$ it is used as an object name. Observe that function symbols can also be applied to path expressions, since path expressions (like id-terms) are used to reference objects. Thus, $f(a.b)$ is legal.

As path expressions and F-logic atoms can be arbitrarily nested, this leads to a concise and very flexible specification language for object properties, as illustrated in the following example.

Example 5.2 (Path Expressions) Consider again the schema given in Figure 1. Given the name n of a person, the following path expression references all editors of conferences in which n had a paper:[9]

_ : conf_p[authors→→{_[name→n]}].at_conf..editors

Therefore, the answer to the *query*

?- P : conf_p[authors→→{_[name→n]}].at_conf[editors→→{E}].

is the set of all pairs (P,E) such that P is (the logical oid of) a paper written by n, and E is the corresponding proceedings editor. If one is also interested in the affiliations of the above editors when the papers were published, we only need to slightly modify our query:

?- P : conf_p[authors→→{_[name→n]}].at_conf[year→Y]..editors[affil@(Y)→A].

Thus, $\mathcal{F}$LORA's path expressions support navigation along the method application dimension using the operators "." and "..". In addition, intermediate objects through which such navigation takes place can be selected by specifying the properties of such objects inside square brackets.

To access intermediate objects that arise implicitly in the middle of a path expression, one can define the method self as X[self→X] and then simply write ... [self→O] ... anywhere in a complex path expression. This would bind the id of the current object to the variable O.[10]

Example 5.3 (Path Expressions with self) Recall the second query in Example 5.2. If the user is also interested in the respective conferences, the query can be reformulated as

X[self→X].
?- P : conf_p[authors→→{_[name→n]}].at_conf[self→C; year→Y]..editors[affil@(Y)→A].

[9] Each occurrence of "_" denotes a distinct don't-care variable (existentially quantified at the innermost level).
[10] A similar feature is used in other languages, e.g., XSQL [2].

5.3 Path Expressions in the Rule Head

Only single-valued path expressions are allowed in a rule head. Set-valued path expressions are not allowed because the semantics is not always clear in such cases.

The following is an example of a path expression in rule head. It says that the mother of person X. The rule defines the grandsons of X's mother.

```
X.mother[grandson->>Y] :- X:person[son->>Y].
```

Complications arise if we specify the following later on:

```
john[mother->mary].
john[son->>david].
```

and ask the following query:

```
?- mary[grandson->>S].
```

Here, we should be able to identify `mary` and `john.mother`, since the attribute `mother` is scalar. To deal with single-valued path expressions in rule heads, $\mathcal{F}$LORA *skolemizes* `john.mother` and derives the requisite equalities. All this is done by the $\mathcal{F}$LORA compiler transparently to the user: if a path expression in rule head is detected, $\mathcal{F}$LORA replaces this expression with a Skolem constant and then appends appropriate rules to the target `.P` file to ensure that proper equalities are maintained.

The user must be aware, however, that *equality maintenance* is costly. Performance can be improved if path expressions in the rule heads are avoided. Our experiments show that without equality checking $\mathcal{F}$LORA can be 10 times faster in some cases.

5.4 References: Truth Value vs. Object Value

Id-terms, F-logic atoms, and path expressions can all be used to reference objects. This is obvious for id-terms and path expressions (7 – 8). Similarly, F-logic atoms (1 – 6) have not only a truth value, but they also reference objects, i.e., yield an object value. For example, $o\!:\!c[m{\rightarrow}r]$ is a reference to o and additionally, it specifies o's membership in class c and the value of the attribute m.

Consequently, all F-logic expressions of the form (1 – 8) are called *references*. F-logic references have a dual reading. Given an F-logic database $\mathcal{I}$ (see below), a reference has:

- An *object value*, which yields the name(s) of the objects reachable in $\mathcal{I}$ by the corresponding expression, and
- A *truth value*, like any other literal or molecule of the language. In particular, a reference r evaluates to *false* if there is no object that is referenced by r in $\mathcal{I}$.

Thus, a path expression can be viewed as a logical formula (*the deductive perspective*), or as an expression that represents one or more objects (*the object-oriented perspective*).

Consider the following path expression and an equivalent (with respect to the truth value) flattening:

$$a..b[c \twoheadrightarrow \{d.e\}] \quad \Leftrightarrow \quad a[b \twoheadrightarrow \{X_{ab}\}] \wedge d[e \rightarrow X_{de}] \wedge X_{ab}[c \twoheadrightarrow \{X_{de}\}]. \qquad (*)$$

Such flattening is used to determine the truth value of arbitrarily complex path expressions in the *body* of a rule. Let `obj (path)` denote the ids of all objects represented by the path expression. Then, for $(*)$, we have:

$$obj(a..b) = \{x_{ab} \mid \mathcal{I} \models a[b \twoheadrightarrow \{x_{ab}\}]\} \qquad \text{and} \qquad obj(d.e) = \{x_{de} \mid \mathcal{I} \models d[e \rightarrow x_{de}]\}\ ,$$

where $\mathcal{I} \models \varphi$ means that φ holds in $\mathcal{I}$. Observe that $obj(d.e)$ contains at most one element because the *single-valued* method e is applied to a single oid d. Thus, two formulas might be equivalent logically, but their values as objects might be different!

In general, for an F-logic database $\mathcal{I}$, the object values of ground expressions are given by the following mapping *obj* from ground references to sets of ground references:

$$\begin{array}{rcl} obj(t) & := & \{t \mid \mathcal{I} \models t[]\},\ \text{for a ground id-term } t \\ obj(o[\ldots]) & := & \{o' \in obj(o) \mid \mathcal{I} \models o'[\ldots]\} \\ obj(o\!:\!c) & := & \{o' \in obj(o) \mid \mathcal{I} \models o'\!:\!c\} \\ obj(c::d) & := & \{c' \in obj(c) \mid \mathcal{I} \models c'::d\} \\ obj(o.m) & := & \{r' \in obj(r) \mid \mathcal{I} \models o[m \rightarrow r]\} \\ obj(o..m) & := & \{r' \in obj(r) \mid \mathcal{I} \models o[m \twoheadrightarrow \{r\}]\} \end{array}$$

Observe that if `t`[] does not occur in $\mathcal{I}$, then $obj(t)$ is $\emptyset$. Conversely, a ground reference r is called *active* if $obj(r)$ is not empty. A reference, r, can be single-valued or multi-valued:

- r is called *multi-valued* if
  - it has the form $o..m$, or
  - it has one of the forms $\underline{o}[\ldots]$, $\underline{o}:c$, $\underline{c}::d$, or $\underline{o}.\underline{m}$, and any of the underlined subexpressions is multi-valued;
- in all other cases, r is *single-valued.*

5.5 Symbols, Strings, Comments

Symbols. $\mathcal{F}$LORA symbols (that are used for the names of constants, predicates, and object constructors) begin with a lowercase letter followed by zero or more letters $(A \ldots Z, a \ldots z)$, digits $(0 \ldots 9)$, or underscores (_), e.g., *student*, *apple_pie*. Symbols can also be *any* sequence of characters enclosed in a pair of single quotes, e.g., `'JOHN SMITH'`, `'default.flr'`. Internally, $\mathcal{F}$LORA symbols are represented as XSB atoms, which are used there as names of predicates and function symbols.

Escaped String	ASCII (decimal)	Symbol
\\	92	\
\n	10	NewLine
\N	10	NewLine
\t	9	Tab
\T	9	Tab
\r	13	Return
\R	13	Return
\v	11	Vertical Tab
\V	11	Vertical Tab
\b	8	Backspace
\B	8	Backspace
\f	12	Form Feed
\F	12	Form Feed
\e	27	Escape
\E	27	Escape
\d	127	Delete
\D	127	Delete
\s	32	Whitespace
\S	32	Whitespace

Table 1: Escaped Character Strings and Their Corresponding Symbols

$\mathcal{F}$LORA also recognizes escaped characters inside single quotes ('). An escaped character normally begins with a backslash (\). Table 1 lists the special escaped character strings and their corresponding special symbols. An escaped character may also be any ASCII character. Such a character is preceded with a backslash together with a lowercase `x` (or an uppercase `X`) followed by one or two hexadecimal symbols representing its ASCII value. For example, `\xd` is the ASCII character Carriage Return, whereas `\x3A` represents the semicolon. In other cases, a backslash is recognized as itself.

One exception is that inside a quoted symbol, a single quote character is escaped by another single quote, e.g., `'isn''t'`.

Strings (character lists). Like XSB strings, $\mathcal{F}$LORA strings are enclosed in a pair of double quotes ("). These strings are represented internally as lists of ASCII characters. For instance, `[102,111,111]` is the same as `"foo"`.

Escape characters are recognized inside $\mathcal{F}$LORA strings similarly to $\mathcal{F}$LORA symbols. However, inside a string, a single quote character does not need to be escaped. A double quote character, however, needs to be escaped by another double quote, e.g., `"""foo"""`.

Numbers. Normal $\mathcal{F}$LORA integers are decimals represented by a sequence of digits, e.g., $892, 12$. $\mathcal{F}$LORA also recognizes integers in other bases (2 through 36). The base is specified by a decimal integer followed by a single quote ('). The digit string immediately follows the single quote. The letters $A \ldots Z$ or $a \ldots z$ are used to represent digits greater than 9. Table 2 lists a few sample integers.

Integer	Base (decimal)	Value (decimal)
1023	10	1023
2'1111111111	2	1023
8'1777	8	1023
16'3FF	16	1023
32'vv	32	1023

Table 2: Representation of Integers

Underscore (_) can be put inside any sequence of digits as delimiters. It is used to partition some long numbers. For instance, `2'11_1111_1111` is the same as `2'1111111111`. However, "_" cannot be the first symbol of an integer, since variables can start with an underscore. For example, 1_2_3 represents the number 123 whereas _12_3 represents a variable named _12_3.

Floating numbers normally look like 24.38. The decimal point must be preceded by an integral part, even if it is 0, i.e., 0.3 must be entered as 0.3, not as .3. Each float may also have an optional exponent. It begins with a lowercase e (or uppercase E) followed by an optional minus sign ($-$) or plus sign ($+$) and an integer. This exponent is recognized as in base 10. For example, `2.43E2=243` whereas `2.43e-2=0.0243`.

Comments. $\mathcal{F}$LORA supports three kinds of comments: (1) all characters following the `%` symbol are interpreted as a comment line; (2) all characters following `//` are also interpreted as a comment line; (3) all characters inside a pairs of `/*` and `*/` are interpreted as comments. Only (3) can span multiple lines.

Note that comments are considered to be white space. Therefore, tokens can also be delimited by comments.

5.6 Aggregation

$\mathcal{F}$LORA uses the same syntax for aggregation as in FLORID. An aggregate looks like this:

agg{X[Gs]; body}

Here, *agg* represents the aggregate operator. X is called the aggregation variable; Gs is a list of comma-separated *grouping* variables. Finally, *body* is a list of literals that specify the conditions. The grouping variables, Gs, are optional.

All the variables appearing in *body* but not in X and Gs are considered to be existentially quantified. Furthermore, the syntax of an aggregate must satisfy the following conditions:
(1) Both X and Gs must appear in *body*; (2) Gs should not contain X.

The following aggregate operators are supported: *min*, *max*, *count*, *sum*, *avg*, *collectset* and *collectbag*.

The operators *min* and *max* can be applied to any list of terms. The order is specified by the XSB operator `@=<`. In contrast, the operators *sum* and *avg* can take numbers only. If the aggregate variable is instantiated to a non-number, *sum* and *avg* will discard it and generate a runtime warning message.

For each group, the operator *collectbag* collects all the bindings of the aggregation variable into a list. The operator *collectset* works similarly to *collectbag*, except that all the duplicates are removed from the result list.

In general, aggregates can appear wherever a number or a list is allowed. Therefore, aggregates can be nested. The following examples illustrate the use of aggregates (some borrowed from the FLORID manual):

?- Z = min{S; john[salary@(Year)→S]}.
?- Z = count{Year; john.salary@(Year) < max{S; john[salary@(Y)→S], Y<Year}}.
?- avg{S[Who]; Who : employee[salary@(Year)→S]} > 20000.

If an aggregate contains grouping variables that are *not* bound by a preceding subgoal, then this aggregate would backtrack over such grouping variables. (In other words, they are considered to be existentially quantified). For instance, in the last query above, the aggregate will backtrack over the variable `Who`. Thus, if `john's` and `mary's` average salary is greater than 20000, this query will backtrack and return both `john` and `mary`.

The following example is a query that for each employee asks for a list of years when this employee had salary less than 60. This illustrates the use of the `collectset` aggregate.

```
?- Z= collectset{Year [Who]; Who[salary@(Year) -> X], X < 60}.
Z = [1990,1991]
Who = mary

Z = [1990,1991,1997]
Who = john
```

5.7 Arithmetic Expressions

Unlike XSB, in $\mathcal{F}$LORA arithmetic expressions are always evaluated (in XSB, + can also be used as a binary functor). Both single-valued and multi-valued path expressions are allowed in arithmetic expressions, and all objects (variables) are considered to be existentially quantified. For example, the following query

?- john..bonus + mary..bonus > 1000.

is actually equivalent to

?- john[bonus↠V1], mary[bonus↠V2], V1 + V2 > 1000.

The only difference is that the values of `V1` and `V2` will be printed out for the latter query, but not for the former one.

Order matters in $\mathcal{F}$LORA. All variables appearing in an arithmetic expression must be instantiated at the time of evaluation. Otherwise, a runtime error will occur.

$\mathcal{F}$LORA allows arithmetic expressions to appear in path expressions. Since arithmetic expressions are always evaluated, an arithmetic expression inside a path expression is treated as the number to which the expression evaluates. Furthermore, $\mathcal{F}$LORA recognizes numbers as oid's, so the result of the evaluation is treated as a regular object.

To illustrate, consider the following example:

```
?- 1.m+2.n.k = X.
```

Since $\mathcal{F}$LORA allows path expressions inside arithmetic expressions, and *vice versa*, it is not immediately obvious whether the previous example stands for the arithmetic expression $(1.m) + (2.n.k)$, or for the path expression $(1.m + 2.n).k$, or $(1.m + 2).n.k$, or $1.(m + 2).n.k$. The correct answer is the first path expression, because "." in a path expression binds stronger than "+" in an arithmetic expression.

One more confusing example is 2.3.4. Does it mean (2).(3).(4), or (2.3).4, or 2.(3.4)? In $\mathcal{F}$LORA, 2.3.4 alone means (2.3).4, since all tokens, like integers, floats, operators, etc., are first processed by $\mathcal{F}$LORA tokenizer and then passed to $\mathcal{F}$LORA parser. In general, the interpretation of "." as a decimal point takes precedence over the interpretation as part of a single-valued path expression.

Another ambiguous situation arises when the symbols $-$ and $+$ are used. Indeed, they can be used as minus/plus signs, e.g., -3 and $+3$, or as binary arithmetic operators; e.g., $4 - 7$ and $4 + 7$. Actually, the minus and plus signs are defined in $\mathcal{F}$LORA as unary operators which take precedence over binary operators.

Table 3 lists various operators in decreasing precedence order, their associativity, and arity.

Wherever ambiguity may arise, parentheses can be used to avoid misleading expressions. Here are more examples of legal expressions in $\mathcal{F}$LORA:

The interpretation of the last expression stems from the fact that both the minus sign and the plus sign are defined as unary operators. Therefore, -6 is a *complex* arithmetic expression (with an arithmetic operator $-$) that represents a method, but not a negative integer.

To avoid further confusion, $\mathcal{F}$LORA insists that all *complex* arithmetic expressions representing oid's in path expressions must be enclosed in parentheses. Thus, although $5.{-6}$ may seem legal according to Table 3, it has to be entered as $5.(-6)$.

5.8 Negation in $\mathcal{F}$LORA

$\mathcal{F}$LORA uses the well-founded semantics for negation and relies on the underlying XSB system for this service. Negation is specified using the `tnot` operator. However, the current implementation has the restriction that `tnot` can be applied only to Prolog predicates, not F-molecules (this restriction will be dropped in a future release). Thus, to negate an F-molecule, one has to introduce

Precedence	Operator	Use	Associativity	Arity
1	()	parentheses	not applied	not applied
2	.	decimal point	not applied	not applied
3	−	minus sign	right	unary
	+	plus sign	right	unary
4	.	path expression	left	binary
5	∗	multiplication	left	binary
	/	division	left	binary
6	−	subtraction	left	binary
	+	addition	left	binary
7	=<	less than or equal to	not applied	binary
	>=	greater than or equal to	not applied	binary
	=:=	equal to	not applied	binary
	=\=	unequal to	not applied	binary
	:=	assignment	not applied	binary
	is	same as :=		

Table 3: Operators in Non-Increasing Precedence Order and Their Associativity and Arity

$(o_1.m_1+o_2.m_2)$.`method`	
2.(3.4)	
$3 + - - 2$	equivalent to $3 + (-(-2))$
$5 * -6$	equivalent to $5 * (-6)$
5.(−6)	method "−6" applied to object "5"

an auxiliary predicate as shown below. Furthermore, this predicate must be tabled (see Section 8):

```
:- table aux/1.
aux(X,Y) :- a[m ->> X; a -> Y].
d[f->Z] :- e[w->Z; v->f(X,Y)], tnot(aux(X,Y)).
```

One other restriction, due to the underlying XSB system, is that all variables in negated predicates must be bound before `tnot` is called.

5.9 Inheritance

F-logic identifies two types of inheritance: *structural* and *behavioral*. Structural inheritance applies to signatures only. For instance, if `student::person` and the program has the signature `person[name ⇒string]` then the query `?- student[name ⇒X]` succeeds with `X = string`.

Behavioral inheritance is much more complex. The problem is that it is *non-monotonic*. That is, addition of new facts might change previously established inferences.

F-logic (and $\mathcal{F}$LORA) distinguishes between attributes and methods that can inherit values from superclasses and those that do not. The syntax that we used so far applies to *non-inheritable* attributes only. *Inheritable attributes* are declared using the `*=>`, `*=>>` style arrows and defined using the `*->`, `*->>` style arrows. For instance, the following is a $\mathcal{F}$LORA program for the classical Royal Elephant example:

```
elephant[color *=> color].
royal_elephant :: elephant.
clyde : elephant.
elephant[color *-> gray].
```

The question is what is the color of Clyde? Clyde's color has not been defined in the above program. However, since Clyde is an elephant and the default color for elephants is gray, Clyde must be gray. Thus, we can derive:

```
clyde[color -> gray].
```

Observe that when inheritable methods are inherited from a class by its members, the attribute becomes non-inheritable. On the other hand, when such a method is inherited by a subclass from its superclass, then the method is still inheritable, so it can be further inherited by the members of that subclass or by its subclasses. For instance, if we have

```
circus_elephant :: elephant.
```

then we can derive

```
circus_elephant[color *-> gray].
```

Non-monotonicity of behavioral inheritance becomes apparent when certain new information gets added to the knowledge base. For instance, suppose that we learn that

```
royal_elephant[color *-> white].
```

Although we have previously established that Clyde is gray, this new information renders our earlier conclusion invalid. Indeed, Since Clyde is a royal elephant, he must be white, while being an elephant he must be gray. The conventional wisdom in object-oriented languages, however, is that inheritance from more specific classes must take precedence. Thus, we must retract our earlier conclusion that Clyde is gray and assume that he is white:

```
clyde[color -> white].
```

Behavioral inheritance in F-logic is discussed at length in [3]. The above problem of non-monotonicity is just a tip of the iceberg. Much more difficult problems arise when inheritance interacts with the regular deduction. To illustrate, consider the following program:

```
b[m *->> c].
a : b.
a[m ->> d] :- a[m ->> c].
```

In the beginning, it seems that `a[m ->> c]` should be derived by inheritance, and so we can derive `a[m ->> d]`. Now, however, we can reason in two different ways:

1. `a[m ->> c]` was derived based on the belief that attribute `m` is not defined for the object `a`. However, once inherited, necessarily we must have `a[m ->> {c,d}]`. So, the value of attribute `m` is not really that produced by inheritance. In other words, inheritance of `a[m ->> c]` negates the very premise on which the original inheritance was based, so we must undo the operation and the ensuing rule application.

2. We did derive `a[m ->> d]` as a result of inheritance, but that's OK — we should not really be looking back and undo previously made inheritance inferences. Thus, the result must be `a[m ->> {c,d}]`.

A semantics that favors the second interpretation was proposed in [3]. This approach is based on a fixpoint computation of non-monotonic behavioral inheritance. However, this semantics is very hard to implement efficiently, especially using a top-down deductive engine provided by XSB. Thus, $\mathcal{F}$LORA uses a different, more cautious semantics for inheritance, which favors the first interpretation above. The idea can be summarized using the following rules, which define how class instances inherit from the classes they belong to. Similar rules are needed to describe how classes inherit from superclasses.

```
// Inheritance rules for scalar attributes
:- table defined/2, overwritten/3.
Obj[A -> V] <- not defined(Obj,A) & Obj:Class & Class[A *-> V]
               & not overwritten(Obj,Class,A) & not conflict(Obj,Class,A).
overwritten(Obj,Class,A) <- Obj:Class1 & Class1::Class
                            & Class1[A *-> W] & Class1 \= Class
defined(Obj,A) <- Obj[A -> V]
conflict(Obj,Class,A) <- defined(Super,A) & Obj:Super
                         & not Super::Class & not Class::Super.

// Inheritance rules for set attributes
:- table definedSet/2, overwrittenSet/3.
Obj[A ->> V] <- not definedSet(Obj,A) & Obj:Class & Class[A *->> V]
                & not overwrittenSet(Obj,Class,A)
                & not conflictSet(Obj,Class,A).
overwrittenSet(Obj,Class,A) <- Obj:Class1 & Class1::Class
                               & Class1[A *->> W] & Class1 \= Class
definedSet(Obj,A) <- Obj[A ->> V]
conflictSet(Obj,Class,A) <- definedSet(Super,A) & Obj:Super
                            & not Super::Class & not Class::Super.
```

Negation here is implemented using the *well-founded semantics* for negation [5, 6] (as indicated by the `tnot` operator).

One problem with the current implementation of behavioral inheritance is that the well-founded semantics for negation in the presence of equality is not yet sufficiently developed. Since $\mathcal{F}$LORA's treatment of inheritance relies on well-founded negation, interaction of equality and inheritance becomes an issue. Fortunately, it is not hard to extend the semantics to the case when derived equalities do not depend on negation or inheritance. In the current implementation of $\mathcal{F}$LORA, it is the responsibility of the programmer to ensure that this is the case: if a derived equality does depend on negation, the result is unpredictable. Interaction between equality and inheritance will be made more structured in a future release.

Inheritable attributes and path expressions. In the previous examples, path expressions used only non-inheritable attributes. Clearly, there is no reason to disallow inheritable attributes in such expressions. To distinguish inheritable from non-inheritable attributes in path expressions, $\mathcal{F}$LORA uses "!" and "!!". For instance,

```
clyde!color          means: some X, such that clyde[color *-> X]
obj!!attr            means: some Y, such that obj[attr *->> Y].
```

5.10 Type Checking

Although $\mathcal{F}$LORA allows the user to specify object types through signatures, type correctness is not being checked automatically. So, what are the signatures good for then? One answer is that future versions of $\mathcal{F}$LORA might support some forms of type checking. However, because F-logic can natively support powerful meta-programming, even the current level of support for signatures is useful. For instance, the programmer can write simple queries to check the types of methods that might look suspicious. Here is one way to construct such a type-checking query:

```
scalar_type_incorrect(O,M,R) :- O[X -> R] , O:C, C[X => D], tnot(R:D).
?- scalar_type_incorrect(obj, meth, Result).
```

Here, we defined what it means to violate type checking using the usual F-logic semantics. The corresponding predicate can then be queried. The "no" answer means that the corresponding attribute *does not* violate the typing rules.

In this way, one can easily consruct special purpose type checkers. This feature is particularly important when dealing with *semi-structured* data. (Semi-structured data has object-like structure but normally does not need to conform to any type; or if it does, the type would normally cover only certain parts of the object structure.)

5.11 Meta-programming in $\mathcal{F}$LORA

The syntax of F-logic lends itself naturally to meta-programming. For instance, it is easy to examine the methods and types defined for the various classes. Here are some simle examples:

```
// All classes where John is a member
?- john : X.
```

```
// All superclasses of student
?- student :: X

// All unary scalar methods defined for object John
?- john[M@(_) -> _].

// All unary scalar methods that apply to John, i.e., for which a
// signature was declared
?- john[M@(_) => _].
```

However, a number of meta-programming primitives are still needed since they cannot be directly expressed in F-logic. Many such features are provided by the underlying XSB system and $\mathcal{F}$LORA simply takes advantage of them:

```
flora ?- functor(X,f,3).
X = f(_h455,_h456,_h457)
Yes.

flora ?- compound(f(X)).
X = _h472
Yes.

flora ?- X =.. [f,a,b].
X = f(a,b)
Yes.
```

These primitives are described in the XSB manual. However, $\mathcal{F}$LORA provides one primitive of its own: a *meta variable* that can range over methods of any arity.

A meta variable is specified by a normal variable immediately prefixed with the "@" sign, *e.g.*, @Method, @_var, @_. Note that @_ represents a *don't care* meta variable. The "@" sign is always considered to be a part of the meta variable's name. Thus, @M and M represent two different variables.

The operator "=.." (similar to that of XSB) is used to obtain a method and its arguments from a meta variable that is bound to a method invocation expression. Alternatively, this operator can be used to *build* a method invocation expression from a list and assign the result to a meta variable. The first element in the list is assumed to represent the method name and the rest represent the arguments. For instance,

```
flora ?- @M =.. [m,a1,a2].
@M = m@(a1,a2)
Yes.
```

The left hand side of "=.." can also be a normal Prolog term. In this case, "=.." acts exactly as in Prolog, *i.e.*, it decomposes the term into a list or constructs a term from a list.

metavar.flr:

:- import length/2 from basics.

$o_1[m_1@(a_1)\rightarrow r_1]$.
$o_1[m_2@(b_1,b_2)\rightarrow r_2]$.
$o_1[m_3\rightarrow r_3]$.
$o_1[m_4@(c_1,c_2,c_3)\rightarrow r_4]$.

o_2[@M→R] :-
    o_1[@M→R],
    @M =.. [Meth|Args],
    length(Args,2).

Figure 2: Using Meta Variables

Consider the example in Figure 2. The rule there "copies" the definitions of methods of arity 1 and 2 from object `o1` to `o2`. To do the same without the meta-variable would require two rules (and more, if we were to copy the methods of higher arities). To see how this works in $\mathcal{F}$LORA, try the following:

```
flora ?- rundemo(metavar).
Yes.

flora ?- o2[@M->R].
@M = m2@(b1,b2)
R = r2
Yes.
```

Currently, a meta variable can appear only where a method invocation is allowed or on the left side of the "`=..`" operator. For instance, `john`[@M→`Salary`], $\mathtt{o}_1$.@M1.$\mathtt{o}_2$[@M2→→`r`]. Thus, unlike the regular variables, meta variables represent method invocations and *not* object. Because of this, you cannot directly pass meta-variables as arguments to predicates and methods. However, you can always convert a meta-variable into a regular variable (*e.g.*, `@M=..N`), pass the regular variable as a parameter, and then convert it back into a meta-variable, as shown below:

```
/* Get some method invocation, convert to normal var, pass on to foo/1 */
?- mary[@Meth -> V], @Meth =.. Param, foo(Param).
/* Convert Param to meta var for method invocations, test object property */
f(Param) :- @M =.. Param, abc[@M ->> 123].
```

6 Compiled Code vs. Dynamic Code

A $\mathcal{F}$LORA program consists of facts and rules all of which take part in the derivation of new facts and object properties. However, there is a distinction between static facts and rules and dynamic ones. The former are immutable, while the latter can be added or deleted at will.

Conceptually, the runtime environment of $\mathcal{F}$LORA is partitioned into two areas: static and dynamic. *Static code* is generated using the predicate `flcompile(file)`, and is loaded into the static runtime environment by `flconsult(file)`, `flload(file)`, or `[file]`. *Dynamic code* can be compiled by `dyncompile(file)` and loaded into the dynamic runtime environment by `dynconsult(file)`, `dynload(file)`, or `<file>`. we have shown the syntax of these predicates in Section 2.

The above predicates can also be called from within a $\mathcal{F}$LORA program, but except for `[file]` and `<file>`, all of them must first be imported from `flrutils` (see Section 7 for details).

Note 1: When a file is compiled and loaded into the dynamic area, all queries that appear in that file are ignored.

Note 2: The same $\mathcal{F}$LORA program can be compiled statically and dynamically, and $\mathcal{F}$LORA puts the two compiled versions into different files. When the program is loaded into the dynamic part of the code, the loader is looking for a dynamically compiled version of the program; when it is loaded into the static part of the code, the loader tries to find a statically compiled version. In particular, it is not possible to load statically what has been compiled dynamically, and vice versa.

Although static and dynamic code resides in different areas, the rules and facts in both these areas are considered as a whole and executed together.

A small example should help illustrate this. Suppose there are two programs, *static.flr* and *dynamic.flr*, as shown in Figure 3. Start XSB in the directory where both *static.flr* and *dynamic.flr* reside. Then start $\mathcal{F}$LORA shell and type:

```
flora ?- flconsult(static).
Yes.

flora ?- dynconsult(dynamic).
Yes.

flora ?- D:department[coursesOffered->>C].

D = cse
C = cse220

D = cse
C = cse310

D = cse
C = cse530
```

static.flr:

department[faculty⇒⇒professor; coursesOffered⇒⇒string].
professor[teaches@(string,number)⇒⇒string].

X : department[coursesOffered→→C] :- X..faculty[teaches@(S,Y)→→C].

cse : department[faculty→→smith].
smith : professor.
smith[teaches@(fall,1998)→→cse220].
smith[teaches@(spring,1999)→→cse310].
smith[teaches@(spring,1999)→→cse530].

dynamic.flr:

math : department[faculty→→john].
john : professor.
john[teaches@(spring,1999)→→math230].
john[teaches@(spring,1999)→→math101].

Figure 3: Static Code vs. Dynamic Code

```
D = math
C = math101

D = math
C = math230
Yes.
```

It can be seen that the two parts of the code work in union. The difference comes when we are trying to modify the code dynamically, *e.g.*, by deleting or adding facts.

$\mathcal{F}$LORA provides the users with several predicates to modify the runtime database. These predicates can be executed either from the static area or the dynamic area. However, only the facts that reside in the dynamic area can be asserted or retracted. (In the furture, $\mathcal{F}$LORA might support asserting and retracting rules in the dynamic area). The database modification predicates supported by $\mathcal{F}$LORA are explained below:

- **assert($P_1, \ldots, P_n$):** asserts a list of facts into the dynamic area. P_i ($i = 1 \ldots n$) can be any F-logic molecule or user defined predicate, e.g.,

`assert(david:professor[teaches@(fall,1999)→→cse505]).`

• `retract`($P_1, \ldots, P_n | C_1, \ldots, C_n$) retracts the *ground* facts corresponding to $P_1, \ldots, P_n$ for which the conjunction of $P_1, \ldots, P_n, C_1, \ldots, C_n$ succeeds. $C_1, \ldots, C_n$ can be considered as the conditions qualifying the facts to be retracted. For instance,

$$\texttt{retract(john[teaches@(S,Y)}\twoheadrightarrow\texttt{C]|smith[teaches@(S,Y)}\twoheadrightarrow\texttt{C])}$$

retracts the teaching information about *john* when it duplicates *smith*'s (*i.e.*, when John and Smith appear to have taught the same course during the same semester). In contrast,

$$\texttt{retract(john[teaches@(S,Y)}\twoheadrightarrow\texttt{C], smith[teaches@(S,Y)}\twoheadrightarrow\texttt{C])}$$

retracts the teaching records of both John and Smith when they duplicate each other.

Special built-in predicates like arithmetic comparison operators cannot be retracted. If P_i happens to be one of those special predicates, $\mathcal{F}$LORA compiler will interpret it as an additional condition C_i and generate a warning. For example,

$$\texttt{retract(john[teaches@(S,Y)}\twoheadrightarrow\texttt{C], Y =< 1999)}$$

is equivalent to

$$\texttt{retract(john[teaches@(S,Y)}\twoheadrightarrow\texttt{C] | Y =< leq1999);}$$

• `retractall`($P_1, \ldots, P_n | C_1, \ldots, C_n$) retracts *all ground* facts corresponding to $P_1, \ldots, P_n$ for which the conjunction of $P_1, \ldots, P_n, C_1, \ldots, C_n$ succeeds. The difference between `retract` and `retractall` is that: `retract` retracts facts one by one and fails if it is unable to retract any facts, whereas `retractall` always succeeds no matter what facts reside in the database. Actually, `retractall` is implemented using `retract` with the following schema (where the arguments are omitted):

```
retractall :- retract, fail.
retractall.
```

• `erase`($P_1, \ldots, P_n | C_1, \ldots, C_n$) retracts *all ground facts* similarly to `retract`. However, in addition, it traces the object reference links and retracts all ground facts referenced along those paths.

To see the effects of `erase`, continue the example of Figure 3:

```
flora ?- erase(cse[faculty->>smith]).
No.
```

Here, $\mathcal{F}$LORA returns "no" because the fact `cse[faculty->>smith]` is located in the static area and thus cannot be retracted.

```
flora ?- erase(math[faculty->>john]).
Yes.
```

Here, the removal of `math[faculty->>john]` proceeds without a hitch, because this fact resides in the dynamic area. More interestingly, all the information about John is also gone as well! This can be seen from the following queries:

```
flora ?- P:professor[teaches@(Semester,Year)->>Course].

P = smith
Semester = fall
Year = 1998
Course = cse220

P = smith
Semester = spring
Year = 1999
Course = cse310

P = smith
Semester = spring
Year = 1999
Course = cse530
Yes.

flora ?- P:professor.
P = smith
Yes.
```

Note that when erasing $O_1 : O_2$ or $O_1 :: O_2$, only the object references that originate from O_1 are followed. For other F-logic facts, such as $O_1[method{\rightarrow}O_2], O_1[method{\twoheadrightarrow}O_2]$, only the object references that originate at Q_2 are followed.

- `eraseall`$(\mathtt{P}_1, \ldots, \mathtt{P}_n | \mathtt{C}_1, \ldots, \mathtt{C}_n)$ erases *all ground* facts corresponding to $\mathtt{P}_1, \ldots, \mathtt{P}_n$ for which the conjunction of $\mathtt{P}_1, \ldots, \mathtt{P}_n, \mathtt{C}_1, \ldots, \mathtt{C}_n$ succeeds. Like `retractall`, `eraseall` always succeeds and is implemented using `erase` via the following schema:

```
eraseall :- erase, fail.
eraseall.
```

Asserting, retracting, and tabling. To implement object properties, $\mathcal{F}$LORA relies on a feature of XSB called *tabling* (see Section 8 for more details). Unfortunately, tabling and `retract` do not mix well. The problem is that results from previous queries are cached in tables, and `retract` does not delete facts from tables. Thus, you might get the following counterintuitive result:

```
flora ?- assert(o[m->1]).
Yes.
flora ?- o[m->1].
Yes.
flora ?- retract(o[m->1]), o[m->1].
Yes.
```

The reason for the wrong positive answer here is that the cache remembers that the query `o[m->1]` is true. So, when the same query is asked after `retract`, a wrong result is returned from the cache. Similarly, tabling might interact poorly with `assert`:

```
flora ?- o[m->1].
No.
flora ?- assert(o[m->1]), o[m->1].
No.
```

The reason for the wrong result here is, again, that the cache remembers that `o[m->1]` is false, which is no longer correct after the assert operation.

In a future release, ℱLORA will provide a workaround for these problems (and it is even possible that a future release of XSB will start doing the right thing in these situations. For now, the only remedy is to use a call to `abolish_all_tables`, which would drop all tables. However, at present, the only safe way to do this is by executing `abolish_all_tables` as a separate query.

7 ℱLORA Modules and Interaction with XSB

Besides static area and dynamic area, ℱLORA also has a module system that is integrated with the XSB's own module system. In this section we discuss how ℱLORA programs can talk with each other and how they can talk to XSB.

Calling ℱLORA from ℱLORA. ℱLORA modules communicate with each other by importing/exporting either *ground* F-logic signatures or normal Prolog predicates. With the rest of XSB, ℱLORA modules communicate using the normal Prolog predicates only (because bare XSB does not speak F-logic).

To illustrate, consider the two ℱLORA modules *module1.flr* and *module2.flr* in Figure 4. Let us start XSB in the directory where both *module1.flr* and *module2.flr* reside, and type the following from the ℱLORA shell:

```
flora ?- flcompile(module2).
... FLORA messages omitted ...
Yes.

flora ?- [module1].
Yes.

flora ?- X=count{Year; john.salary@(Year) < mary.salary@(Year)}.
X = 2
Yes.
```

What you see here is that `module1` is loaded and the query is posed. However, `module1` only contains information about John. Mary's information is kept in `module2`. However, `module1`

module1.flr:

:- import employee[salary@(number)⇒number] from module2.

john : employee.
john[salary@(1994)→70].
john[salary@(1995)→80].
john[salary@(1996)→70].
john[salary@(1997)→50].
john[salary@(1998)→80].

module2.flr:

:- export employee[salary@(number)⇒number].

employee[salary@(number)⇒number].
mary : employee.
mary[salary@(1994)→60].
mary[salary@(1995)→60].
mary[salary@(1996)→70].
mary[salary@(1997)→80].
mary[salary@(1998)→90].

Figure 4: Example $\mathcal{F}$LORA Modules

imports this information from `module2`, and the imported information takes part in the query evaluation process.

The import/export directives can take a list of predicate/arity pairs (as XSB does) and/or *ground* F-logic signatures (no variables are allowed in the signatures that are imported or exported). For example,

$$:-\texttt{import tc/2, student[grade@(string)}\Rightarrow\texttt{number], p/1 from foo.}$$

is allowed, but

$$:-\texttt{import student[G@(string)}\Rightarrow\texttt{number] from foo.}$$

is not.

When a $\mathcal{F}$LORA module imports from another module, say, module `foo.flr`, the latter must already be compiled, or else a runtime error will be issued. Furthermore, a $\mathcal{F}$LORA module can not import and export the same signature. These restrictions result from the limitations of the underlying XSB module system.

Import directives can appear in both static and dynamic code. However, all export directives (as well as queries, as mentioned earlier) are ignored when a $\mathcal{F}$LORA module is compiled as dynamic code and/or is dynamically loaded into the dynamic area. This is, again, due to the current limitations of the XSB module system.

mix.flr:

```
:- import findall/3 from setof.

edge(a,b).
edge(b,c).
edge(c,b).

string[reachableTo⇒⇒string].

X : activeNode[reachableTo→→Y] :- edge(X,Y).
X : activeNode[reachableTo→→Y] :- edge(X,Z), Z[reachableTo→→Y].

tc(X,Y) :- X[reachableTo→→Y].

show(X) :-
        X : activeNode,
        write(X),
        write('[reachableTo→→{'),
        findall(Y,tc(X,Y),L),
        writelist(L),
        writeln('}]').

writelist([X]) :- write(X).
writelist([X1, X2|Xs]) :- write(X1), write(','), writelist([X2|Xs]).
```

Figure 5: Mixing $\mathcal{F}$LORA code with XSB code

Calling XSB from $\mathcal{F}$LORA. Since $\mathcal{F}$LORA supports import/export directives much the same way as XSB does, $\mathcal{F}$LORA modules have full access to the underlying XSB's functionality. In general, a $\mathcal{F}$LORA program can call any XSB predicate that is exported by some XSB module. This is done by importing this predicate in the $\mathcal{F}$LORA program.

$\mathcal{F}$LORA programs can freely mix F-logic statements and XSB predicates defined in other XSB modules as long as these XSB predicates are properly imported and are used correctly.

Consider the example in Figure 5 and suppose that the following queries are entered at the $\mathcal{F}$LORA prompt:

```
flora ?- [mix].
Yes.

flora ?- show(a), show(b).
a[reachableTo->>{b,c}]
b[reachableTo->>{b,c}]
```

```
Yes.
```

Observe that in Figure 5 we created a new predicate, `tc`, and used *it* as an argument to `findall` (which is a standard Prolog predicate; see the XSB manual). It seems more natural to write `findall(Y,X[reachableTo->>Y],L)` instead. This more natural syntax will be supported in the future, but it does not work at the present time. The reason is that $\mathcal{F}$LORA compiler always treats F-logic molecules as oid's, if they appear as predicate arguments. However, in `findall`, we want the molecules in the second argument to be treated as logical formulas that evaluate to true or false. This will be supported in a future release via a special compiler directive.

Since $\mathcal{F}$LORA can use most of the services provided by XSB, reading the XSB manual is highly recommended in order to be productive. Some services, such as I/O, are of obvious importance. However, there are many other useful packages, which provide pattern matching capabilities, interaction with the OS, foreign C interface, etc.

Calling $\mathcal{F}$LORA from XSB. Programs written in $\mathcal{F}$LORA can be used by XSB program as well. Of course, XSB does not understand $\mathcal{F}$LORA syntax directly, but they share the same common denominator: Prolog predicates. Thus, a $\mathcal{F}$LORA module can define a predicate, export it, and XSB programs can then import and call it. Full power of F-logic syntax can be used in such a definition. The predicate syntax is needed only at the final stage, to create a communication channel to XSB.

XSB programs can even compile and consult $\mathcal{F}$LORA programs. To this end, they must have the import statements of the following form:

```
:- import bootstrap_flora/0 from flora.
:- import flcompile/1 from flrutils.
:- import flconsult/1 from flrutils.
?- bootstrap_flora.
```

The statement `bootstrap_flora` is a non-interactive equivalent of the command `flora_shell`. It makes all the $\mathcal{F}$LORA facilities available without actually starting the shell (which is what one really wants while calling $\mathcal{F}$LORA programs from other programs). Once the `bootstrap_flora` statement has been executed, we can call, say, `flconsult(foobar)` from within XSB programs to compile (if necessary) and load the $\mathcal{F}$LORA program `foobar.flr`.

8 $\mathcal{F}$LORA and Tabling

Tabling is a technique that enhances top-down evaluation with a mechanism that remembers the calls previously made during query evaluation. This technique is known to be essentially equivalent to the Magic Sets method for bottom-up evaluation. However, tabling combined with top-down evaluation has the advantage of being able to utilize highly optimized compilation techniques developed for Prolog. The result is a very efficient deductive engine.

XSB lets the user specify which predicates must be tabled. The FLORA compiler automatically tables the predicates used to flatten F-logic molecules. However, the user is responsible for telling the system which other predicates must be tabled. (Normally, these are predicates defined by the user.) FLORA programs accepts the same tabling directives as XSB does (Section 9 lists all the compiler directives).

It is important to keep in mind that XSB does not do reordering of objects and predicates during joins. Instead, all joins are performed left-to-right. The programmer, thus, must write program clauses in such a way as to ensure that smaller predicates and classes appear early on in the join. Also, even though XSB tables the results obtained from previous queries, the current tabling engine has several limitations. In particular, when a new query comes in, XSB tries to determine if this query is "similar" to one that already has been answered (or is in the process of being evaluated). Unfortunately, the current notion of similarity used by XSB is fairly weak, and many unnecessary recomputations might result. This problem will be corrected in a future release.

It is also important to be aware that when XSB (and FLORA) evaluate a program, all tabled predicates are partially materialized and all the computed tuples are stored in XSB tables. Thus, if you change the set of facts, the existing tables must be discarded in order to allow XSB to recompute the results. This is accomplished by issuing the predicate `abolish_all_tables/0` described in the XSB manual.

Furthermore, tabling sometimes has undesirable side effects in "real-world" programming, especially when writing methods with non-logical side effects (*e.g.*, writing or reading a file). If a tabled predicate has such side effects, then the first time the predicate is called the side effect will be performed, but the second time the call simply returns with success or failure (depending on the outcome of the first call). Thus, if the predicate was intended to perform the side effect each time it is called, it will not operate correctly.

All this is, of course, old news to XSB programmers, but is there anything FLORA-specific in this? It turns out that yes, and the problem is not immediately apparent. In the object-oriented style, people tend to define methods with side effects and attach them to objects. However, because FLORA tables everything that comes from F-molecules, methods with side effects are subject to the same problem as described above. The current interim solution is to use predicates instead of methods whenever side effects are needed. In a future release, FLORA will have special syntax for methods with side effects, so this restriction will be lifted.

No discussion of a logic programming language is complete without a few words about the infamous Prolog cut (!). Although Prolog cut has been (mostly rightfully) excommunicated by as far as Database Query Languages are concerned, it is sometimes indispensable when doing "real work", like pretty-printing FLORA programs or implementing a pattern matching algorithm. To facilitate this kind of tasks, FLORA lets the programmer use cuts. However, the current implementation of XSB has a limitation that Prolog cuts cannot "cut across tabled predicates." Without trying to pretend to be experts, we refer the reader to the XSB manual for details on this obscure problem. The XSB team is considering correcting this problem in a future release.

For now, enjoy your cut. If you get an error message telling something about cutting across the tables — you know that you may have cut too much :-). The basic rule that can keep you out of trouble is: do not put a cut in the body of a rule *after* any F-molecule. However, it is (usually)

OK to to put a cut before any F-molecule. It is even OK to have a cut in the body of a rule that *defines* an F-molecule (again, provided that the body has no F-molecule to the left of that cut).

9 $\mathcal{F}$LORA Compiler

Like XSB compiler, $\mathcal{F}$LORA compiler can take compilation directives. All such directives must begin with `:-` (while all queries must begin with `?-`). The following is a list of all the compiler directives supported by $\mathcal{F}$LORA:

Tabling Directive Tabling directive can be either "`:- auto_table.`", which lets XSB automatically decide which predicates should be tabled, or "`:- table p_a_list.`", where `p_a_list` is a coma-separated list of predicate/arity pairs specifying those predicates to be tabled. Note that the tabling directive is needed only for the user-defined predicates inside $\mathcal{F}$LORA modules. The internal $\mathcal{F}$LORA predicates that are used to implement F-logic atoms are tabled automatically.

Import Directive Import directive is of the form "`:- import sig_p_a_list.`", where `sig_p_a_list` is a list of *ground* F-logic signatures and/or predicate/arity pairs.

Export Directive Export directive is of the form "`:- export sig_pa_list.`", where *sig_p_a_list* is a list of *ground* F-logic signatures and/or predicate/arity pairs. All export directives are ignored if a $\mathcal{F}$LORA module is compiled as dynamic code and/or is loaded dynamically.

Equality Maintenance Directive Equality maintenance directive has the form "`:- eqlevel(N).`" where the level number N specifies the degree to which $\mathcal{F}$LORA will try to maintain the equalities among objects derived during query evaluation. Currently, only two levels of equality maintenance are supported: 0 (no equality maintenance) and 1 (full equality maintenance).

Equality maintenance directives can appear in several places in a $\mathcal{F}$LORA program. However, if `eqlevel(1)` is requested somewhere in the module, $\mathcal{F}$LORA will compile the module with equality maintenance level 1.

Note that equality level 1 should not be specified unnecessarily, since it can slow $\mathcal{F}$LORA down by an order of magnitude. The default equality maintenance level is 0. However, if $\mathcal{F}$LORA compiler detects a path expression in a rule head, which requires Skolemization (which, for correctness, requires full equality maintenance), it automatically switches to the equality level 1. Therefore, path expressions in the rule head must be avoided if at all possible.

Equality maintenance can also be requested when $\mathcal{F}$LORA modules are compiled using predicates such as `flcompile` and `flconsult` (which must be imported from the module `flrutils`). For instance, `flcompile(benchmark,[eqlevel(1)])` will compile `benchmark.flr` with equality maintenance level 1. If equality maintenance is given both in the `flcompile` command and inside the flora module being compiled, the highest level will be selected by the compiler.

Here is the full list of compilation and loading predicates, all imported from `flrutils`, that can be used in conjunction with FLORA:

`flcompile(File, Directives)`	–	compile File with compilation Directives.
`flcompile(File)`	–	same with default directives.
`flconsult(File, Directives)`	–	consult File with compilation Directives.
`flconsult(File)`	–	same with default directives.
`dyncompile(File, Directives)`	–	like flcompile/2, but compiles as FLORA dynamic code.
`dyncompile(File)`	–	same with default directives.
`dynconsult(File, Directives)`	–	like flconsult/2, but consults as FLORA dynamic code.
`dynconsult(File)`	–	same with default directives.
`flload(File)`	–	load File.{flr,P,O} as static code.
`dynload(File)`	–	load File.{flr,P,O} as dynamic code.

10 FLORA Debugger

FLORA debugger is essentially a presentation layer on top of the XSB debugger, so familiarity with the latter is highly recommended (XSB Manual, Part I). Here we sketch only a few basics.

The debugger has two facilities: tracing and spying. Tracing allows the user to watch the program being executed step by step, and spying allows one to tell FLORA that it must pose when execution reaches certain predicates or object methods. The user can trace the execution from then on. At present, only the tracing facility has been implemented.

To start tracing, you must issue the command `flora_trace` at the FLORA prompt. It is also possible to put the subgoal `flora_trace` in the middle of the program. In tat case, tracing will start after this subgoal gets executed. This is useful when you know where exactly you want to start tracing the program. To stop tracing, type `flora_notrace`.

During tracing, the user is normally prompted at the four ports of subgoal execution: `Call` (when a subgoal is first called), `Exit` (when the call exits), `Redo` (when the subgoal is tried with a different binding on backtracking), and `Fail` (when a subgoal fails). At each of the prompts, the user can issue a number of commands. The most common ones are listed below. See the XSB manual for more.

- `carriage return (creep)`: to go to the next step
- `s (skip)`: execute this subgoal non-interactively; prompt again when the call exits (or fails)
- `S (verbose skip)`: like `s`, but also show the trace generated by this execution
- `l (leap)`: stop tracing and execute the remainder of the program

The behavior of the debugger is controled by the predicate `debug_ctl`. For instance, executing `debug_ctl(profile, on)` at the FLORA prompt tells XSB to measure the CPU time it takes to execute each call. This is useful for tuning your program for performance. Other useful controls are: `debug_ctl(prompt, off)`, which causes the trace to be generated without user intervention; and `debug_ctl(redirect, foobar)`, which redirects debugger output to the file named `foobar`. The latter feature is usually useful only in conjunction with the aforesaid prompt-off mode. See the XSB manual for additional information on debugger control.

11 Emacs Support

Editing and debugging $\mathcal{F}$LORA programs can be greatly simplified with the help of *flora-mode*, a special Emacs editing mode designed specifically for $\mathcal{F}$LORA programs. Flora-mode provides support for syntactic highlighting, automatic indentation, and the ability to run $\mathcal{F}$LORA programs right out of the Emacs buffer.

11.1 Instalation

To install *flora-mode*, you must perform the following steps. Put the file

```
XSB/packages/flora/emacs/flora.el
```

found in your XSB distribution on the load path of Emacs or XEmacs (whichever you are using). The best way to work with Emacs is to make a separate directory for Emacs libraries (if you do not have one), and put `flora.el` there. Such a directory can be added to emacs search path by putting the following command in the file `~/.emacs` (or `~/.xemacs`, if you are running one of the newer versions of XEmacs):

```
(setq load-path (cons "your-directory" load-path))
```

It is also a good idea to compile emacs libraries. To compile flora.el, use this:

```
emacs -batch -f batch-byte-compile flora.el
```

If you are using XEmacs, use `xemacs` instead of `emacs` above — the two emacsen often use incompatible byte code.

Finally, you must tell X/Emacs how to recognize $\mathcal{F}$LORA program files, so Emacs will be able to invoke the Flora major mode automatically when you are editing such files:

```
(setq auto-mode-alist (cons '("\\.flr$" . flora-mode) auto-mode-alist))
(autoload 'flora-mode "flora" "Major mode for editing Flora programs." t)
```

To enable syntactic highlighting of Emacs buffers (not just for $\mathcal{F}$LORA programs), you can do the following:

- In Emacs: select `Help.Options.Global Font Lock` on the menubar. To enable highlingting permanently, put

```
(global-font-lock-mode t)
```

in `~/.emacs`.

- In XEmacs: select `Options.Syntax Highlighting.Automatic` in the menubar. To enable this permanently, put

```
(add-hook 'find-file-hooks 'turn-on-font-lock)
```

in `~/.emacs` or `~/.xemacs` (whichever is used by your XEmacs).

11.2 Functionality

Menubar menu. Once flora editing mode is installed, it provides a number of functions. First, whenever you edit a $\mathcal{F}$LORA program, you will see the "Flora" menu in the menubar. This menu provides commands for controlling the Flora process (*i.e.*, XSB with the $\mathcal{F}$LORA shell). You can start and stop this process, type queries to it, and you can tell it to consult regions of the buffer you are editing, the entire buffer, or some other file.

Because Emacs provides automatic file completion and allows you to edit what you typed, performing these functions right out of the buffer takes much less effort than typing the corresponding commands on XSB command line.

Keyboard functions. In addition to the menu, *flora-mode* lets you execute most of the menu commands using the keyboard. Once you get the hang of it, keyboard commands are much faster to invoke:

```
Consult file:                 Ctl-c Ctl-f
Consult file dynamically:     Ctl-u Ctl-c Ctl-f
Consult buffer:               Ctl-c Ctl-b
Consult buffer dynamically:   Ctl-u Ctl-c Ctl-b
Consult region:               Ctl-c Ctl-r
Consult region dynamically:   Ctl-u Ctl-c Ctl-r
```

When you invoke any of the above commands, a $\mathcal{F}$LORA process is started, unless it is already running. However, if you want to invoke this process explicitly, type

```
ESC x run-flora
```

You can control the $\mathcal{F}$LORA process using the following commands:

```
Interrupt Flora Process:      Ctl-c Ctl-c
Quit Flora Process:           Ctl-c Ctl-d
Restart Flora Process:        Ctl-c Ctl-s
```

Interrupting $\mathcal{F}$LORA is equivalent to typing `Ctl-c` at the $\mathcal{F}$LORA prompt, quitting the process stops XSB, and restarting the process shuts down the old XSB process and starts a new one with $\mathcal{F}$LORA shell running.

Indentation. Flora editing mode understands some aspects of the FLORA syntax, which enables it to provide correct indentation of program lines (in many cases). In the future, flora mode will know more about the syntax, which will let it provide even better support for indentation.

The most common use of FLORA indentation facility is by typing the `TAB`-key. If *flora-mode* manages to understand where the cursor is, it will indent the line accordingly. Another way is to put the following in your emacs startup file (`~/.emacs` or `~/.xemacs`):

```
(setq flora-electric t)
```

In this case, whenever you type the return key, the next line will be indented automatically.

12 Future Enhancements

FLORA is work in progress. We are still experimenting with features and nothing is cast in stone. So, although we do not intend to make the life of FLORA users harder than it already is, we cannot give a guarantee of backward compatibility. The following enhancements and features are among currently planned:

Syntax enhancements: At present, the `not` operator can be applied to predicates only. This restriction will be removed in the future, so it will be possible to negate arbitrary F-molecules.

A future version of FLORA will support a no-op. So, it will be possible to write F-molecules without worrying about the semicolon, *i.e.*, `a[m->b;;c->d;]`. It will be also possible to have extraneous commas: `head :- b1,,b2,.`

Equality: In a future release, equality maintenance will most likely be more restrictive: only the objects that are explicitly equated via an equality predicate in the head of a rule will be allowed to be equated. Any other derived equality will be treated as an error (or a very stern warning).

Aggregates: FLORA will provide builtin functions that will directly apply to the outcome of the aggregates `collectbag` and `collectset`. This will make it possible to do grouping once and then compute multiple aggregates over the groups.

Work areas: At present, FLORA supports only one dynamic work area. A future version of FLORA will support multiple dynamic work areas.

If-then-else: FLORA will provide the equivalent of the if-then-else construct, to make programs more readable.

Transaction logic: FLORA will be enhanced with Transaction Logic syntax.

Additional compiler directives: An F-molecule and a path expression have two meanings: as an oid and as a truth value. Currently, an F-molecule or a path expression that occurs inside a predicate is interpreted as an object. This is not always desirable, however. For instance, in `findall`, it is more appropriate to evaluate F-molecules that occur in the second argument to truth values. This can be done with compiler directives like:

```
:- arguments findall(oid,truth,oid)
```

meaning that the first and the third arguments should be evaluated to their oids and the second argument should be evaluated to a truth value.

$\mathcal{F}$LORA will also allow database declarations, like those used in the XSB database interface.

References

[1] J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *VLDB*, pages 273–284, 1994.

[2] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 393–402, New York, June 1992. ACM.

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

[4] 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), 1998.

[5] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *ACM Principles of Database Systems*, pages 1–10, New York, 1989. ACM.

[6] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Index