$\mathcal{F}$LORA-*2*: User's Manual

Version 0.95

(Androcymbium)

Guizhen Yang[1] Michael Kifer[2] Hui Wan[2] Chang Zhao[2]

[1]SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, U.S.A.

[2]Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.

October 2, 2007

Contents

1 Introduction

$\mathcal{F}$LORA-*2* is a sophisticated object-oriented knowledge base language and application development platform. It is implemented as a set of run-time libraries and a compiler that translates a unified language of F-logic [8], HiLog [4], and Transaction Logic [2, 1] into tabled Prolog code.

Applications of $\mathcal{F}$LORA-*2* include intelligent agents, Semantic Web, ontology management, integration of information, and others.

The programming language supported by $\mathcal{F}$LORA-*2* is a dialect of F-logic with numerous extensions, which include a natural way to do meta-programming in the style of HiLog and logical updates in the style of Transaction Logic. $\mathcal{F}$LORA-*2* was designed with extensibility and flexibility in mind, and it provides strong support for modular software design through its unique feature of dynamic modules. Other extensions, such as the versatile syntax of FLORID path expressions, are borrowed from FLORID, a C++-based F-logic system developed at Freiburg University.[1] Extensions aside, the syntax of $\mathcal{F}$LORA-*2* differs in many important ways from FLORID, from the original version of F-logic, as described in [8], and from an earlier implementation of $\mathcal{F}$LORA. These syntactic changes were needed in order to bring the syntax of $\mathcal{F}$LORA-*2* closer to that of Prolog and make it possible to include simple Prolog programs into $\mathcal{F}$LORA-*2* programs without choking the compiler. Other syntactic deviations from the original F-logic syntax are a direct consequence of the added support for HiLog, which obviates the need for the "@" sign in method invocations (this sign is now used to denote calls to $\mathcal{F}$LORA-*2* modules).

$\mathcal{F}$LORA-*2* is available on $\mathcal{F}$LORA-*2*'s Web site at `http://flora.sourceforge.net`

Installing $\mathcal{F}$LORA-*2* in UNIX. To install the latest release of $\mathcal{F}$LORA-*2* or its current development version, download it from `http://flora.sourceforge.net` into a separate directory *outside* the XSB installation tree. After unpacking (or checking out from CVS) the $\mathcal{F}$LORA-*2* sources will be placed in the `flora2` subdirectory of the current directory. To configure $\mathcal{F}$LORA-*2*, do the following:

```
cd flora2
make clean
./makeflora
```

(assuming that XSB has been already installed and configured). If an XSB executable is not on your program search `PATH`, then in the third command above you need to provide the XSB installation directory to `makeflora` as an argument, *e.g.*,

```
./makeflora all ~/XSB
```

if XSB is installed in the directory ∼`/XSB`.

Installing $\mathcal{F}$LORA-*2* in Windows. First, you need Microsoft's `nmake`, which can be downloaded from `http://download.microsoft.com/download/vc15/Patch/1.52/W95/EN-US/Nmake15.exe`. The

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

file `Nmake15.exe` is a self-extracting archive; when it is run, it extracts the program files for `nmake.exe` into the same directory where `Nmake15.exe` resides. Let's say this directory is `C:\Nmake`. To unpack the $\mathcal{F}$LORA-*2* file archive, you also need WinZIP.

Once `nmake.exe` is installed and the $\mathcal{F}$LORA-*2* archive is unpacked, use the following commands to configure $\mathcal{F}$LORA-*2* (assuming that XSB is already installed and configured):

```
PATH=C:\Nmake;%PATH%     ( assuming that nmake.exe  is in C:\Nmake)
cd directory-where-you-unpacked-flora2
makeflora clean
makeflora path-to-prolog-executable
```

Here *directory-where-you-unpacked-flora2* is the directory where you unpacked $\mathcal{F}$LORA-*2*; it should have the form *something*`\flora2`. The *path-to-prolog-executable* must be the full path name of the XSB executable. Note, that unlike Unix, there should be no "all" after "makeflora" in Windows.

It is also recommended that you set the environment variable `HOME` on your Windows system, if it is not already defined. Environment variables are usually set by opening the `System` folder, which is located inside the `Settings` folder. Typically, the `HOME` variable is set to the directory

```
"C:\Documents and Settings\your-user-name"
```

This can prevent problems with upgrading to the latest version of $\mathcal{F}$LORA-*2*.

If you are a developer and wish to recompile the C part of $\mathcal{F}$LORA-*2* (and provided you have a Microsoft C++ compiler), then you can type[2]

```
makeflora -c path-to-prolog-executable
```

Normally, however, there is no need to do so.

Installing $\mathcal{F}$LORA-*2* in Windows under Cygwin. Although $\mathcal{F}$LORA-*2* runs under native Windows, it runs faster under Cygwin, because the underlying Prolog engine has special optimizations for GCC.

To install $\mathcal{F}$LORA-*2* under Cygwin, configure XSB as in Unix and use the default options:

```
cd XSB/build
./configure
./makexsb
```

Then change to the $\mathcal{F}$LORA-*2* directory and configure $\mathcal{F}$LORA-*2*:

[2] A version of this compiler (which is all you need in order to compile XSB or $\mathcal{F}$LORA-*2*) can be downloaded free of charge from `http://msdn.microsoft.com/vstudio/express/visualC/default.aspx` It is also necessary to install Windows Platform SDK accessible from the above page.

```
cd directory-where-you-unpacked-flora2
make clean
./makeflora all path-to-prolog-executable
```

If XSB can be found through the PATH environment variable then you can simply type `./makeflora`.

Running $\mathcal{F}$LORA-*2*. $\mathcal{F}$LORA-*2* is fully integrated into the underlying Prolog engine, including its module system. In particular, $\mathcal{F}$LORA-*2* modules can invoke predicates defined in other Prolog modules, and Prolog modules can query the objects defined in $\mathcal{F}$LORA-*2* modules. At present, XSB is the only Prolog platform where $\mathcal{F}$LORA-*2* can run, because it heavily relies on tabling and the well-founded semantics for negation, both of which are available only in XSB.

Due to certain problems with XSB, $\mathcal{F}$LORA-*2* runs best when XSB is configured with *local* scheduling, which is the default XSB configuration. However, with this type of scheduling, many Prolog intuitions that relate to the operational semantics do not work. Thus, the programmer must think "more declaratively" and, in particular, to not rely on the order in which answers are returned.

The easiest way to get a feel of the system is to start $\mathcal{F}$LORA-*2* shell and begin to enter queries interactively. The simplest way to do this is to use the shell script

```
.../flora2/runflora
```

where "..." is the directory where $\mathcal{F}$LORA-*2* is downloaded. For instance,

```
~/FLORA/flora2/runflora
```

At this point, $\mathcal{F}$LORA-*2* takes over and F-logic syntax becomes the norm. To get back to the Prolog command loop, type `Control-D` (Unix) or Control-Z (Windows), or

```
flora2 ?- _end.
```

If you are using $\mathcal{F}$LORA-*2* shell frequently, it pays to define an alias, say (in Bash):

```
alias flora2='xsb -e "[flora2], flora_shell."'
alias runflora='~/FLORA/flora2/runflora'
```

$\mathcal{F}$LORA-*2* can then be invoked directly from the shell prompt by typing `flora2` or `runflora`. It is even possible to tell $\mathcal{F}$LORA-*2* to execute commands on start-up. For instance,

```
foo>  flora2 -e "\_end."
foo>  runflora -e "\_end."
```

will cause the system to execute the help command right after after the initialization. Then the usual $\mathcal{F}$LORA-*2* shell prompt is displayed.

$\mathcal{F}$LORA-*2* comes with a number of demo programs that live in

```
.../flora2/demos/
```

The demos can be run issuing the command "`_demo(demo-filename).`" at the FLORA-2 prompt, *e.g.*,

```
flora2 ?- _demo(flogic_basics).
```

There is no need to change to the demo directory, as `_demo` knows where to find these programs.

2 FLORA-2 Shell Commands

Loading programs from files. The most common shell command you probably need are the commands for loading and compiling a program:

flora2 ?- `[programfile].`

or

flora2 ?- `_load(programfile).`

Here `program-file` can contain a FLORA-2 program or a Prolog program. If `program-file.flr` exists, it is assumed to be a FLORA-2 program. The system will compile the program, if necessary, and then load it. The compilation process is two-stage: first, the program is compiled into a Prolog program (one or more files with extensions `.P` and `.fdb`) and then into an executable byte-code, which has the extension `.xwam`.

If there is no `program-file.flr` file, the file is assumed to contain a Prolog program and the system will look for the file named `program-file.P`. This file then is compiled into `program-file.xwam` and loaded. Note that in this case the program is loaded into a *Prolog module* of FLORA-2 and, therefore, calls to the predicates defined in that program must use the appropriate module attribution — see Section 11.1 for the details about the module system in FLORA-2.

By default, all FLORA-2 programs are loaded into the module called `main`, but you can also load programs into other modules using the following command:

```
flora2 ?- [file>>modulename].
```

Understanding FLORA-2 modules is very important in order to be able to take full advantage of the system; we will discuss the module system of FLORA-2 in Section 11.1. Once the program is loaded, you can pose queries and invoke methods for the objects defined in the program.

There is an important special case of the `_load` and `[...]` command when the file name is `_` (underscore). In that case, instead of looking for the program file `_.flr`, FLORA-2 starts reading user input. At this point, the user can start typing in program clauses, which the system saves in a temporary file. When the user is done and types the end of file character `Control-D` (Unix) or `Control-Z` (Windows), the file is compiled and loaded. It is also possible to load such a program into a designated module, rather than the default one, using one of the following commands:

```
flora2 ?- [file>>module].
flora2 ?- _load(file>>module).
```

Adding rule-bases to modules. When the `_load` command loads a rule base into a module, it first wipes out all the rules and facts that previously formed the knowledge base of that module. Sometimes it is desirable to *add* the facts and rules contained in a certain file to the already existing knowledge base of a module. This operation, called `_add`, does not erase the old knowledge base in the module in question. It is also possible to use the `[...]` syntax by prefixing the file name with a `+`-sign. Here are some examples of adding a rule-base contained in files to existing modules:

```
flora2 ?- [+foo].
flora2 ?- [+foo>>bar].
flora2 ?- _add(foo).
flora2 ?- _add(foo>>bar).
```

When using the `[...]` syntax, adding and loading can be intermixed. For instance,

```
flora2 ?- [foo>>bar, +foo2>>bar].
```

This first loads the file `foo.flr` into the module `bar` and then adds the rule base contained in `foo2.flr` to the same module.

Reporting answers to queries. When the user types in a query to the shell, the query is evaluated and the results are returned. A result is a tuple of values for each variable mentioned in the query, except for the *anonymous variables* represented as "`?_`" or ?, and named *don't care variables*, which are preceded with the underscore, *e.g.*, `?_abc`.

By default, FLORA-2 prints out all answers. If only one at a time is desired, type in the following command: `_one`. You can revert back to the all-answers mode by typing `_all`. Note: `_one` and `_all` affect only the *subsequent* queries. That is, in

```
flora2 ?- \_one, goallist1.
flora2 ?- goallist2.
```

the `_one` directive will affect `goallist2`, but *not* `goallist1`. This is because `goallist1` executes in the same query as `_one` and thus is not affected by this directive.

FLORA-2 shell includes many more commands beyond those mentioned above. These commands are listed below. However, at this point the purpose of some of these commands might seem a bit cryptic, so it is a good idea to come back here after you become more familiar with the various concepts underlying the system.

Summary of shell commands. In the following command list, the suffixes `.flr .P`, `.xwam` are optional. If the file suffix is specified explicitly, the system uses the file with the given name without any modification. The `.flr` suffix denotes a FLORA-2 program, the `.P` suffix indicates that it is a

Prolog program, and `.xwam` means that it is a bytecode file, which can be executed by Prolog. If no suffix is given, the system assumes it is dealing with a FLORA-2 program and adds the suffix `.flr`. If the file with such a name does not exist, it assumes that the file contains a Prolog program and tries the suffix `.P`. Otherwise, it tries `.xwam` in the hope that an executable Prolog bytecode exists. If none of these tries are successful, an error is reported.

- `_end`: Show the help info.
- `_compile(file)`: Compile FILE.flr for the default module `main`.
- `_compile(file>>module)`: Compile FILE.flr for the module `module`.
- `_load(file>>module)`: Load `file.flr` into the module `module`. If you specify `file.P` or `file.xwam` then will load these files.
- `_load(file)`: Load `file.flr` into the default module `main`. If you specify `file.P` or `file.xwam` then will load these files.
- `_compile(file)`: Compile FILE.flr for *adding* to the default module `main`.
- `_compileadd(file>>module)`: Compile FILE.flr for adding the module `module`.
- `_add(file>>module)`: Add `file.flr` to the module `module`.
- `_add(file)`: Add `file.flr` to the default module `main`.
- `[file.{P|xwam|flr} >> module,...]`: Load the files in the specified list into the module `module`. The files can optionally be prefixed with a "+", which means that the file should be added to the module rather than loaded into it.
- `_demo(demofilename)`: Consult a demo from FLORA-2 demos directory.
- `abolish_all_tables`: Flush all tabled data. This is sometimes needed when Prolog's tabling gets in the way. We describe tabling (as it pertains to FLORA-2) in Section 17.
- `_op(Precedence,Associativity,Operator)`: Define an operator in shell mode.
- `_all`: Show all solutions (default). Affects subsequent queries only.
- `_one`: Show solutions to subsequent queries one by one.
- `_trace/_notrace`: Turn on/off FLORA-2 trace.
- `_chatter/_nochatter`: Turn on/off the display of the number of solutions at the end of query evaluation.
- `_end`: Say Ciao to FLORA-2, stay in Prolog.
- `_halt`: Quit both FLORA-2 and Prolog.

Of course, many other executable directives and queries can be executed at FLORA-2 shell. These are described further in this manual

In general, FLORA-2 built-in predicates whose name is of the form `fl[A-Z]...` are either the FLORA-2 shell commands or predicates that can be used in Prolog to control the execution of FLORA-2 modules. We will discuss the latter in Section 11.8. Some of these commands — mostly dealing with loading and compilation of FLORA-2 modules — can also be useful within FLORA-2 applications.

All commands with a FILE argument passed to them use the Prolog `library_directory` predicate to search for the file, except that the command `_demo(FILE)` first looks for `FILE` in the FLORA-2 demo directory. The search path typically includes the standard system's directories used by Prolog followed by the current directory.

All Prolog commands can be executed from FLORA-2 shell, if the corresponding Prolog library has already been loaded.

After a parsing or compilation error, FLORA-2 shell will discard tokens read from the current input stream until the end of file or a rule delimiter (".") is encountered. If FLORA-2 shell seems to be hanging after the message

```
++FLORA Warning: discarding tokens (rule delimiter '.' or EOF expected)
```

hit the `Enter` key once, type ".", and then `Enter` again. This should reset the current input buffer and you should see the FLORA-2 command prompt:

```
flora2 ?-
```

3 F-logic and FLORA-2 by Example

In the future, this section will contain a number of small introductory examples illustrating the use of F-logic and FLORA-2. Meanwhile, the reader can read the FLORA-2 tutorial, which is available on the FLORA-2 Web site: `http://flora.sourceforge.net/tutorial.php`.

Other tutorials exist for systems that use F-logic as their knowledge representation language. A tutorial for the FLORID project is at `http://www.informatik.uni-freiburg.de/~dbis/florid/`. A tutorial for Ontobroker, a commercial system from Ontoprise.de, can be found at `http://www.ontoprise.de/documents/tutorial_flogic.pdf`. FLORA-2 shares much of the syntax with those other systems with the following notable differences: FLORA-2 uses "," as the separator between methods in object formulas, while these other systems use ";". In addition, FLORA-2 does not use the @-sign between method names and arguments.

4 Differences Between FLORA-2 Syntax and F-logic Syntax

FLORA-2 was developed several years after the publication of the initial works on F-logic [8] and so it benefits from the experience gained in the use and implementation of the logic. This experience

led us to introduce some changes to the syntax (and to some degree also to the semantics). The main differences are enumerated below.

1. FLORA-2 uses "," to separate methods in F-molecules. The version of the logic in [8] used ";". In FLORA-2, ";" represents disjunction instead. It is also possible to use "and" instead of "," and "or" instead of ";".

2. FLORA-2 does not use the @-sign to separate method names from their arguments. With HiLog extensions the "@" sign is redundant.

3. `p::p` is not a tautology in FLORA-2, i.e., "::" is not reflexive. This is because our experience showed that the non-reflexive use of "::" is a more common idiom in knowledge representation.

4. In [8], types are always inheritable, but values are not. For instance, the property `a[b->c]` is not inheritable to the subclasses of `a`, but the property `a[b*->c]` is. In FLORA-2, the notation for types is brought in line with the notation for values. In particular, FLORA-2 uses `*=>` for inheritable types and `=>` for non-inheritable ones. The original F-logic in [8] used only `=>` (and `=>>` because it distinguished between functional and set-valued methods), and both were inheritable.

   The semantics of `*=>` are characterized by the following inference rules:

   ```
   X[M *=> T], Y::X ⊨ Y[M *=> T]
   X[M *=> T], Y:X ⊨ Y[M => T]
   ```

   which is analogous to the behavior of `->`.

5. The type inference rules for input restriction and output relaxation introduced in [8] are not implemented in FLORA-2.

6. The syntax `a[b=>{c,d}]` of F-logic, which states that the type returned by the attribute `b` is the intersection of the classes `c` and `d`, is not allowed. Use `a[b=>(c,d)]` instead. FLORA-2 also allows `a[b=>(c;d)]`, `a[b=>(c-d)]` and combinations of these operators on types.

7. Instead of `class[method => {}]` one should use `class[method => ()]`.

8. Equality (the `:=:` predicate) is implemented only partially in FLORA-2. The main limitation is that the congruence axiom for equality ("substitution by equals") works only at the top level and the first level of nesting. For deeper levels of nesting, substitution by equals has not been implemented. This is discussed in more detail in Section 15.1.

9. Behavioral inheritance has a different (and better) semantics in FLORA-2 compared to [8]. This is discussed in Section 14.

5 Basic FLORA-2 Syntax

In this section we describe the basic syntactic structures used to build FLORA-2 programs. Subsequent sections describe the various advanced features that are needed to build practical applications.

The complete syntax is given in Appendix A. However, it should be noted that BNF cannot describe the syntax of FLORA precisely, because it is based on operator grammar (like in Prolog) mixed with context free grammars in places where operator grammar is inadequate (as, for example, in parsing if-then-else).

5.1 F-logic Vocabulary

- *Symbols*: The F-logic alphabet of *object constructors* consists of the sets $\mathcal{C}$ and $\mathcal{V}$ (variables). Variables are symbols that begin with a questionmark, followed by a letter or an underscore, and then followed by zero or more letters and/or digits and/or underscores (e.g., ?X, ?name, ?_, ?_v_5). All other symbols, including the constants (which are 0-ary object constructors), are symbols that start with a letter followed by zero or more letters and/or digits and/or underscores (e.g., a, John, v_10). They are called *constant symbols*. Constant symbols can also be any string of symbols enclosed in single quotes (e.g., 'AB@*c'). Later, in Section 25, we introduce additional constants, called typed literals.

  In addition to the usual first-order connectives and symbols, there is a number of special symbols:], [, }, {, ",", ";", %, #, _#, ->, =>, :, ::, ->->, *->->, :=:, etc.

- *Anonymous and don't care variables*: Variables of the form ?_ or ? are called *anonymous* variable. It is used whenever a *unique* new variable is needed. In particular, two different occurrences of ?_ or ? in the same clause are treated as *different* variables. Named variables that start with an underscore, e.g., ?_foo, are called *don't care* variables. Unlike anonymous variables, two different occurrences of such a variable in the same clause refer to the *same* variable. Nevertheless, don't variables have special status when it comes to error checking and returning answers. The practice of logic programming shows that a singleton occurrence of a variable in a clause is often a mistake due to misspelling. Therefore, FLORA-2 issues a warning when it finds that some variable is mentioned only once in a clause. If such an occurrence is truly intended, it must be replaced by an anonymous variable or a don't care variable to avoid the warning message from FLORA-2. Also, bindings for anonymous and don't care variables are not returned as answers.

- *Id-Terms/Oids*: Instead of the regular first-order terms used in Prolog, FLORA-2 uses HiLog terms. HiLog terms [4] generalize first-order terms by allowing variables in the position of function symbols and even other terms can serve as functors. For instance, p(a)(?X(f,b)) is a legal HiLog term. Formally, a HiLog term is a constant, a variable, or an expression of the form $\mathtt{t}(\mathtt{t}_1, ..., \mathtt{t}_n)$ where t, t_1, ..., t_n is a HiLog term.

  HiLog terms over $\mathcal{C}$ 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*. Numbers (including integers and floats) can also be used as Id-terms, but such use might be confusing and is not recommended.

- *Atomic formulas*: Let $\mathtt{O}, \mathtt{M}, \mathtt{R}_\mathtt{i}, \mathtt{X}_\mathtt{i}, \mathtt{C}, \mathtt{D}, \mathtt{T}$ be Id-terms. In addition to the usual first-order atomic formulas, like $p(X_1, \ldots, X_n)$, there are the following basic types of formulas:

  1. O[M -> V], O[M *-> V]

2. O[M -> {V_1, ..., V_n}], O[M *-> {V_1, ..., V_n}]
3. C[M => T], C[M *=> T]

In all of the above cases, O, C, M, V_i, and T_i are HiLog terms, *i.e.*, expressions of the form, a, f(?X), ?X(s, ?Y), ?X(f, ?Y)(?X, g(k)), etc., where ?X and ?Y are variables and lowercase letters f, s, etc., are constants.

Expressions (1) and (2) above are *data atoms* for *value-returning* methods. They specify that a *method expression* M applied to an object O returns the result object V in case (1), or a set of objects, V_1, ..., V_n, in case (2). In all cases, methods are assumed to be set-valued. However, later we will see that cardinality constraints can be imposed on methods, so it would be possible to state that a particular method is functional or has some other cardinality property. The formula (2) says that the result consists of several objects, which *includes* V_1, V_2, ..., V_n. Note that we emphasized "includes" to make it plain that other facts and rules in the knowledge base can specify additional objects that must be included among the method result.

When $n = 1$ in (2), the curly braces can be omitted. For instance, O[M -> V_1]. In fact, the single expression (2) is equivalent to a the following set of expressions, where the result set is split into singletons:

```
O[M -> V1]
O[M -> V2]
  ...
O[M -> Vn]
```

When M is a constant, *e.g.*, abc, then we say that it is an *attribute*; for example, John[name -> 'John']. When M has the form f(X,Y,Z) then we refer to it as a method, f, with arguments X, Y, and Z; for example, John[salary(1998) -> 50000]. However, as we saw earlier, method expressions can be much more general than these two possibilities, as they can be arbitrary HiLog terms.

The expression (3) above is a *signature atom*. It specifies a *type constraint*, which says that the method expression, M, when applied to objects that belong to class C, must yield objects that belong to class T.

Note: FLORA-2 does not automatically enforce type constraints. However, run-time type checking is possible—see Section 26.2.

Objects are grouped into classes using *ISA-atoms*:

4. O : C
5. C :: D

The expression (4) states that O is an *instance* of class C, while (5) states that C is a *subclass* of D.

User-defined equality

6. `O1 :=: O2`

enables the user to state that two syntactically different (and typically non-unifiable) terms represent the same object. For instance, one can assert that `a :=: b` and from then on everything that is true about `a` will be true about `b`, and vice versa. Note that this is different and more powerful than the unification-based equality builtin `=`, which exists both in FLORA-2 and Prolog. For instance, `=`-based formulas can never occur as a fact or in the rule head, and `a = b` is always false. More on user-defined equality in Section 15.1.

- *F-molecules* provide a convenient way to shortcut specifications related to the same object. For instance, the conjunction of the atoms `John:person`, `John[age->31]`, `John[children ->{Bob,Mary}]`, and `John[children->John]` is equivalent to the following single F-molecule:

  ```
  John:person[age->31, children->{Bob,Mary,John}]
  ```

  Note the use of the "," that separates the expression for the `age` attribute from the expression for the `children` attribute. This is a departure from the original F-logic syntax in [8], which uses ";" to separate such expressions.

- *Rules* are, as usual, the constructs of the form $head :- body$, where *head* is an F-molecule and *body* is a conjunction of F-molecules or negated F-molecules. (Negation is specified using `\+` or `not`— the difference will be explained later.) Each rule must be terminated with a ".".

  Conjunction is specified as in Prolog, using the "," symbol. Like in Prolog, FLORA-2 also allows disjunction in the rule body, which is denoted using ";". As usual in logic languages, a single rule of the form

  $$\begin{array}{ll} head \ :- & \texttt{John[age->31]}, \\ & (\texttt{John[children->\{Bob,Mary\}]} \ ; \ \texttt{John[children->John]}). \end{array} \quad (1)$$

  is equivalent to the following pair of rules:

  ```
  head :- John[age->31], John[children->{Bob,Mary}].
  head :- John[age->31], John[children->John].
  ```

  Disjunction is also allowed inside F-molecules. For instance, the rule (1) can be equivalently rewritten as:

  ```
  head :- John[age->31, (children->{Bob,Mary} ; children->John)].
  ```

  Note that conjunction "," binds stronger than disjunction ";", so the parentheses in the above example are essential.

- *Programs and queries*: A *program* is a set of rules. A *query* is a rule without the head. In FLORA-2, such headless rules use `?-` instead of `:-`, *e.g.*,

  ```
  ?- John[age->?X].
  ```

  The symbol `:-` in headless FLORA-2 expressions is used for various directives, which are plenty and will be introduced in due course.

Example 5.1 (Publications Database) Figure 1 depicts a fragment of a FLORA-2 program that represents a database of scientific publications.

Schema:
conf_p :: paper.
journal_p :: paper.
paper[authors => person, title => string].
journal_p[in_vol => volume].
conf_p[at_conf => conf_proc].
journal_vol[of => journal, volume => integer, number => integer, year => integer].
journal[name => string, publisher => string, editors => person].
conf_proc[of_conf => conf_series, year => integer, editors => person].
conf_series[name => string].
publisher[name => string].
person[name => string, affil(integer) => institution].
institution[name => string, address => string].

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

Figure 1: A Publications Object Base and its Schema in FLORA-2

5.2 Symbols, Strings, and Comments

Symbols. FLORA-2 symbols (that are used for the names of constants, predicates, and object constructors) begin with a letter followed by zero or more letters (`A...Z, a...z`), digits (`0...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, FLORA-2 symbols are represented as *Prolog symbols*,[3] which are used there as names of predicates and function symbols. All FLORA-2 symbols belong to the class `_symbol`.

FLORA-2 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

[3] Symbols are called "atoms" in Prolog, which contravenes the use of this term for atomic formulas in classical logic and F-logic. We avoid the use of the term "atom" in reference to 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

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.

If it is necessary to include a single quote inside a quoted symbol, that single quote must be escaped by another single quote, e.g., `'isn''t'` or by a backslash, e.g., `'isn\'t'`.

Character lists. Like Prolog character lists, FLORA-2 character lists (*charlists*) are enclosed in a pair of double quotes (`"`). For instance, `[102,111,111]` is the same as `"foo"`.

Escape characters are recognized inside FLORA-2 charlists similarly to FLORA-2 symbols. However, inside a charlist, 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"""`. or by a backslash.

Numbers. Normal FLORA-2 integers are decimals represented by a sequence of digits, e.g., `892`, `12`. FLORA-2 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...Z` or `a...z` are used to represent digits greater than 9. Table 2 lists a few example 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, "_"

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

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, e.g., `0.3` must be entered as `0.3`, but not as `.3`. Each floating number may also have an optional exponent. It begins with a lowercase `e` or an 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 is 243` whereas `2.43e-2 is 0.0243`.

Other data types. FLORA-2 supports an array of primitive data types, including string, Boolean, dateTime, iri, and more. Primitive data types are described in Section 25.

Comments. FLORA-2 supports two kinds of comments: (1) all characters following `//` until the end of the line; (2) all characters inside a pair of `/*` and `*/`. Note that only (2) can span multiple lines.

Comments are recognized like whitespaces by the compiler. Therefore, tokens can also be delimited by comments.

5.3 Operators

As in Prolog, FLORA-2 allows the user to define operators, to liven up the syntax. There are three kinds of operators: infix, prefix, and postfix. An infix operator appears between its two arguments, while a prefix operator before its single argument and a postfix operator after its single argument. For instance, if `foo` is defined as an infix operator, then `?X foo a` will be parsed as `foo(?X,a)` and if `bar` is a postfix operator then `?X bar` is parsed as `bar(?X)`.

Each operator has a *precedence level*, which is a positive integer. Each operator also has a *type*. The possible types for infix operators are: `xfx`, `xfy`, `yfx`; the possible types for prefix operators are: `fx`, `fy`; and the possible types for postfix operators are: `xf`, `yf`. In each of these expressions, `f` stands for the operator, and `x` and `y` stand for the arguments. The symbol `x` in an operator expression means that the precedence level of the corresponding argument should be *strictly less* than that of the operator, while `y` means that the precedence level of the corresponding argument should be *less or equal* than that of the operator.

The precedence level and the type together determine the way the operators are parsed. The general rule is that precedence of a constant or a functor symbol that has not been defined as an

operator is zero. Precedence of a Prolog term is the same as the precedence of its main functor. An expression that contains several operators is parsed in such a way that the operator with the highest precedence level becomes the main functor of the parsed term, the operator with the next-highest precedence level becomes the main functor of one of the arguments, and so on. If an expression cannot be parsed according to this rule, a parse error is reported.

It is not our goal to cover the use of operators in any detail, since this information can be found in any book on Prolog. Here we just give an example that illustrates the main points. For example, in FLORA-2, `-` has precedence level `800` and type `yfx`, `*` has precedence level `700` and type `yfx`, `->` has precedence level 1100 and type `xfx`. Therefore, `8-2-3*4` is the same as `-(-(8,2),*(3,4))` in prefix notation, and `a -> b -> c` will generate a parsing error.

Any symbol can be defined as an operator. The general syntax is

```
:- _op(Precedence,Type,Name).
```

For instance,

```
:- _op(800, xfx, foo)
```

As a notational convenience, the argument `Name` can also be a list of operator names of the same type and precedence level, for instance,

```
:- _op(800,yfx,[+,-]).
```

It is possible to have more than one operator with the same name provided they have different use (*e.g.*, one infix and the other postfix). However, the FLORA-2 built-in operators are not allowed to be redefined. In particular, any symbol that is part of F-logic syntax, such as ",', ".", "[", ":", etc., as well as any name that begins with `flora` or `fl` followed by a capital letter should be considered as reserved for internal use.

Although this simple rule is sufficient, in most cases, to keep you out of trouble, you should be aware of the fact that symbols such as "`,`", "`;`", "`+`", "`.`", "`->`", "`::`", "`:-`", "`?-`" and many other parts of FLORA-2 syntax are operators. Therefore, there is a chance that precedence levels chosen for the user-defined operators conflict with those of FLORA-2 and, as a result, your program might not parse. If in doubt, check the declarations in the file `flroperator.P` in the FLORA-2 source code.

The fact that some symbols are operators can sometimes lead to surprises. For instance,

```
?- (a,b,c).
:- (a,b).
```

will be interpreted as terms `'?-'(a,b,c)` and `':-'(a,b)` rather than a query and a directive, respectively. The reason for this is that, first, such terms are allowed in Prolog and there is no good reason to ban them in FLORA-2; and, second, the above syntax is ambiguous and the parser makes the choice that is consistent with the choice made in Prolog. Typically users do not put parentheses around subgoals in such cases, and would instead write

```
?- a,b,c.
:- a,b.
```

Note that things like

```
?- (a,b),c.
?- ((a,b,c)).
```

will be interpreted as queries, so there are plenty of ways to satisfy one's fondness for redundant parentheses.

5.4 Logical Expressions

In a FLORA-2 program, any combination of conjunction, disjunction, and negation of literals can appear wherever a logical formula is allowed, e.g., in a rule body.

Conjunction is represented through the infix operator "`,`" and disjunction is made using the infix operator "`;`". Negation is made through the prefix operators "`\+`" and "`not`".[4] When parentheses are omitted, conjunction binds stronger than disjunction and the negation operators bind their arguments stronger than the other logical operators. For example, in FLORA-2 the following expression: `a, b; c, not d`, is equivalent to the the logical formula: $(\mathtt{a} \wedge \mathtt{b}) \vee (\mathtt{c} \wedge (\neg \mathtt{d}))$.

Logical formulas can also appear inside the specification of an object. For instance, the following F-molecule:

```
o[not att1->val1, att2->val2; meth->res]
```

is equivalent to the following formula:

```
(not o[att1->val1], o[att2->val2]) ; o[meth->res]
```

5.5 Arithmetic Expressions

In FLORA-2 arithmetic expressions are *not* always evaluated. As in Prolog, the arithmetic operators such as `+`, `-`, `/`, and `*`, are defined as normal binary functors. To evaluate an arithmetic expression, FLORA-2 provides another operator, `is`. For example, `?X is 3+4` will bind `?X` to the value `7`.

When dealing with arithmetic expressions, the order of literals is important. In particular, all variables appearing in an arithmetic expression must be instantiated at the time of evaluation. Otherwise, a runtime error will occur. For instance,

```
?- ?X > 1, ?X is 1+1.
```

[4] In brief, "`\+`" represents negation as failure and can be applied only to non-tabled Prolog, FLORA-2, or HiLog predicates. "`not`", on the other hand, is negation that implements the well-founded semantics. Refer to Section 13 for more information on the difference between negation operators.

will produce an error, while

```
?- ?X is 1+1, ?X > 1.
```

will evaluate to true.

As in Prolog, the operands of an arithmetic expression can be any variable or a constant. However, in FLORA-2, an operand can also be a *path expression*. For the purpose of this discussion, a path expression of the form `p.q` should be understood as a shortcut for `p[q->?X]`, where `?X` is a new variable, and `p.q.r` is a shortcut for `p[q->?X], ?X[r->?Y]`. More detailed discussion of path expressions appears in Section 7.

In arithmetic expressions, and all variables are considered to be existentially quantified. For example, the following query

```
flora2 ?- John.bonus + Mary.bonus > 1000.
```

should be understood as

```
flora2 ?- John[bonus->?_V1], Mary[bonus->?_V2], ?_V1 + ?_V2 > 1000.
```

Note that in first query does not have any variables, so after the evaluation the system would print either yes or no. To achieve the same behavior, we use *don't care variables*, `?_V1` and `?_V2`. If we used `?V1` and `?V2` instead, the values of these variables would have been printed out.

FLORA-2 recognizes numbers as oids and, thus, it is perfectly normal to have allows arithmetic expressions inside path expressions such as this: `1.2.(3+4*2).7`. When parentheses are omitted, this might lead to ambiguity. For instance, is the meaning of

```
1.m+2.n.k
```

represented by the arithmetic expression `(1.m)+(2.n.k)`, or by the path expressions `(1.m+2.n).k`, by `(1.m + 2).n.k`, or by `1.(m+2).n.k`? To disambiguate such expressions, we must remember that the operator "." used in path expressions binds stronger than the arithmetic operators $+$, $-$, etc.

Even more interesting is the following example: `2.3.4`. Does it represent the path expression `(2).(3).(4)`, or `(2.3).4`, or `2.(3.4)` (where in the latter two cases 2.3 and 3.4 are interpreted as decimal numbers)? The answer to this puzzle (according to FLORA-2 conventions) is `(2.3).4`: when tokenizing, FLORA-2 first tries to classify tokens into meaningful categories. Thus, when 2.3 is first found, it is identified as a decimal. Thus, the parser receives the expression (2.3).4, which it identifies as a path expression that consists of two components, the oids 2.3 and 4.

Another ambiguous situation arises when the symbols `-` and `+` are used as minus and plus signs, respectively. FLORA-2 follows the common arithmetic interpretation of such expressions, where the `+/-` signs bind stronger than the infix operators and thus `4--7` and `4-+7` are interpreted as `4-(-7)` and `4-(+7)`, respectively.

Table 3 lists various operators in decreasing precedence order, their associativity, and arity. When in doubt, use parentheses. Here are some more examples of valid arithmetic expressions:

Precedence	Operator	Use	Associativity	Arity
not applicable	()	parentheses	not applicable	not applicable
not applicable	.	decimal point	not applicable	not applicable
	.	object reference	left	binary
	:	ISA specification	left	binary
	::	subclass specification	left	binary
600	-	minus sign	right	unary
	+	plus sign	right	unary
700	*	multiplication	left	binary
	/	division	left	binary
800	-	subtraction	left	binary
	+	addition	left	binary
	=<	less than or equals to	not applicable	binary
	>=	greater than or equals to	not applicable	binary
1000	=:=	equals to	not applicable	binary
	=\=	unequal to	not applicable	binary
	is	assignment	not applicable	binary

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

`o1.m1+o2.m2.m3`	same as `(o1.m1)+(o2.m2)`
`2.(3.4)`	the value of the attribute `3.4` on object `2`
`3 + - - 2`	same as `3+(-(-2))`
`5 * - 6`	same as `5*(-6)`
`5.(-6)`	the value of the attribute `-6` on object `5`

Note that the parentheses in `5.(-6)` are needed, because otherwise "`.-`" would be recognized as a single token. Similarly, the whitespace around "`+`", "`-`", and "`*`" are also needed in these examples to avoid `*-` and `+--` being interpreted as distinct token.

6 Class Expressions

*F*LORA-*2* defines a number of set-theoretic operations on classes. For instance, (a, b) represents *intersection*, $(a; b)$ represents the union, and $(a - b)$ represents the difference between the extensions of class a and b. Suppose the following information is given:

```
a, b, c  in class1
c        in class2
e        in class3
```

Then $(\texttt{class1} - \texttt{class2}); \texttt{class3}$ has the extension of $\texttt{a}, \texttt{b}, \texttt{e}$.

We call the above combinations of types **class expressions**. Type expressions can occur in signature expressions as shown below:

```
cl[attr => ((c1 - c2) ; c3)].
cl[attr *=> ((c1,c2) ; c3)].
```

Note that the old F-logic syntax `a[b=>{c,d}]` for type intersection (of `c` and `d`) is no longer permitted.

$\mathcal{F}$LORA-*2* also defines a number of subclass relationships among class expressions as follows.

1. If `c::c1` and `c::c2` then `c::(c1,c2)`, i.e., `(c1,c2)` is the least upper bound of `c1` and `c2` in the class hierarchy.
2. If `c1::c` and `c2::c` then `(c1;c2)::c`, i.e., `(c1;c2)` is the greatest lower bound of `c1` and `c2` in the class hierarchy.
3. Any class, `c`, is considered a superclass of `(c,?_)` and `(?_,c)`. In particular, `(c,c)::c`. At present, $\mathcal{F}$LORA-*2* does not enforce the equality `c:=:(c,c)`.
4. Any class, `c`, is considered a subtype of `(c;?_)` and `(?_;c)`. In particular, `c::(c;c)`. At present, $\mathcal{F}$LORA-*2* does not enforce the equality `c:=:(c;c)`.
5. Any class, `c`, is considered a superclass of `c-d` for any class `d`.

Unfortunately, these subclass relationships can adversely affect certain user programs and $\mathcal{F}$LORA-*2* provides an optimization option that allows the user to disable these relationships for programs that do not need them. See Section 27.2.

Note: Type expressions introduce a potential for infinite answers for seemingly innocuous queries. For instance, suppose that `a:c` is true. Then also `a:(c,c)`, `a:(c;c)`, `a:(c,(c,c))`, `a:(c;(c;c))`, etc. So, the query `?- a:?X.` will not terminate. To mitigate this problem, when class expressions are involved $\mathcal{F}$LORA-*2* guarantees to provide sound answers to queries about class membership and subclasses only when the arguments are ground; it does not guarantee that all class expressions will be returned to queries that involve open calls to "::" and ":".

7 Path Expressions

In addition to the basic F-logic syntax, the $\mathcal{F}$LORA-*2* system also supports *path expressions* to simplify object navigation along value-returning method applications, and to avoid explicit join conditions [6]. The basic idea is to allow the following *path expressions* wherever Id-terms are allowed:

7. `O.M`

Path expressions are allowed *only in rule bodies*. The path expression in (7) refers to the unique object $\mathtt{R_0}$ for which $\mathtt{O[M->R_0]}$ holds. The symbols `O` and `M` stand for an Id-term or path a expression. Moreover, `M` can be a method that takes arguments, in which case $\mathtt{O.M(P_1,\ldots,P_k)}$ is a valid path expressions.

In order to disambiguate the syntax and to specify the desired order 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, which specifies the object o such that a[b -> x] ∧ x[c -> o] holds (note that x = a.b). In contrast, a.(b.c) is the object o1 such that b[c -> x1] ∧ a[x1 -> o1] holds (note that in this case, x1 = b.c). In general, o and o1 can be different objects. Note also that in (a.b).c, b is a method name, whereas in a.(b.c) it is used as an object name and b.c as a method. Observe that function symbols can also be applied to path expressions, since path expressions, like Id-terms, represent objects. Thus, f(a.b) is a valid expression.

Note: A path expression can appear wherever an oid can, but not in place of a truth-valued expression (*e.g.*, a subquery) even though path expressions can be viewed as formulas. Thus,

```
?- ?P.authors.
```

is illegal and will cause a compiler error. To use a path expression as a query, square brackets must be attached. For instance, the following are legal queries:

```
?- ?P.authors[].
?- ?P.authors[name->?N].
```

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

Example 7.1 (Path Expressions) Consider again the schema given in Figure 1. If n is the name of a person, the following path expression is a query that returns all editors of conferences in which n had a paper:

```
flora2 ?- ?P[authors -> {?[name -> n]}].at_conf.editors[].
```

Likewise, the answer to the query

```
flora2 ?- ?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 we also want to see the affiliations of the above editors, we only need to modify our query slightly:

```
flora2 ?- ?P[authors -> {?[name -> n]}].at_conf[year -> ?Y].editors[affil(?Y) -> ?A].
```

Thus, $\mathcal{F}$LORA-*2* path expressions support navigation along the method application dimension using the operator ".". In addition, intermediate objects through which such navigation takes place can be selected by specifying the properties of such objects inside square brackets.[5]

To access intermediate objects that arise implicitly in the middle of a path expression, one can define the method self as

```
?X[self -> ?X].
```

[5] A similar feature is used in other languages, e.g., XSQL [7].

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`.

Example 7.2 (Path Expressions with self) To illustrate convenience afforded by the use of the `self` attribute in path expressions, consider the second query in Example 7.1. If, in addition, we want to obtain the names of the conferences where the respective papers were published, that query can be reformulated as follows:

```
?X[self -> ?X].
?- ?P[authors -> ?[name -> n]].at_conf[self -> ?C,year -> ?Y].editors[affil(?Y) -> ?A].
```

8 Truth Values and Object Values

Id-terms, F-logic atoms, and path expressions can all be used as objects. This is obvious for Id-terms and the object interpretation of path expressions of the form (7) and (8) on page 19 was discussed through 10 are typically viewed as formulas and, thus, they are assumed to have a truth value only. However, there also is a natural way to give them object interpretation. For example, `o:c[m->r]` has object value `o` and some truth value. However, unlike the object value, the truth value depends on the database (on whether `o` belongs to class `c` in the database and whether the value of the attribute `m` is, indeed, `r`.

Although previously we discussed only the object interpretation for path expressions, it is easy to see that they have truth values as well, because a path expression corresponds to a conjunction of F-logic atoms. Consequently, all F-molecules of the form (1) through (7) have dual reading: As logical formulas (*the deductive perspective*), and as expressions that represent one or more objects (*the object-oriented perspective*). Given an intended model, $\mathcal{I}$, of an F-logic program an expression has:

- An *object value*, which yields the Id(s) of the object(s) that are reachable in $\mathcal{I}$ by the corresponding expression, and
- A *truth value*, like any other literal or molecule of the language.

An important property that relates the above interpretations is: a molecule, r, evaluates to *false* if $\mathcal{I}$ has no object corresponding to r.

Consider the following path expression and an equivalent, decomposed expression:

$$\mathtt{a.b[c->\{d.e\}]} \quad \Leftrightarrow \quad \mathtt{a[b->?X_{ab}]} \wedge \mathtt{d[e->?X_{de}]} \wedge \mathtt{?X_{ab}[c->?X_{de}]}. \tag{2}$$

Such decomposition 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 (2) above, we have:

$$obj(\mathtt{d.e}) = \{\mathtt{x_{de}} \mid \mathcal{I} \models \mathtt{d[e->x_{de}]}\}$$

where $\mathcal{I} \models \varphi$ means that φ holds in $\mathcal{I}$. Observe two formulas can be equivalent, but their object values might be different. For instance, `d[e->f]` is equivalent to `d.e` as a formula. However, *obj*(`d.e`) is f, while *obj*(`d[e->f]`) is `d`.

In general, for an F-logic database $\mathcal{I}$, the object values of ground path expressions are given by the following mapping, *obj*, from ground molecules to sets of ground oids (t, o, c, d, m can be oids or path expressions):

$$\begin{array}{rcl} obj(t) & := & \{t \mid \mathcal{I} \models t[]\}, \text{ for a ground Id-term } t \\ obj(o[\ldots]) & := & \{o1 \mid o1 \in obj(o), \mathcal{I} \models o1[\ldots]\} \\ obj(o:c) & := & \{o1 \mid o1 \in obj(o), \mathcal{I} \models o1:c\} \\ obj(c::d) & := & \{c1 \mid c1 \in obj(c), \mathcal{I} \models c1::d\} \\ obj(o.m) & := & \{r1 \mid r1 \in obj(r), \mathcal{I} \models o[m\texttt{->}r]\} \end{array}$$

Observe that if `t[]` does not occur in $\mathcal{I}$, then *obj*(t) is $\emptyset$. Conversely, a ground molecule r is called *active* if *obj*(r) is not empty.

Dual representation and meta-predicates. Since path expressions can appear wherever Id-terms are allowed, the question arises whether a path expression is intended to indicate a truth value or an object value. For instance, we may want to call a predicate `foobar/1`, which expects as an argument a formula because the predicate calls this formula as part of the definition. For instance, the predicate may take a formula and a variable that occurs in that formula and joins this formula with some predicate using that variable:

```
foobar(?Form,?Var) :- ?Form, mypred(?Var).
?- foobar(a[b->?X], ?X).
```

If all arguments are treated as objects, then the above query would mean

```
?- a[b->?X], foobar(a,?X).
```

and an unintended result will be obtained.

The problem here is that the interpretation of F-logic expressions as objects is not always what we want. In our example, we need to indicate to the compiler that the first argument of `foobar/1` ought to be translated into Prolog as follows:

```
foobar(𝒫,?X)
```

where $\mathcal{P}$ is the object that represents the formula `a[b->?X]` *itself* rather than just the oid `a`.

This can be accomplished using the *reification* feature of $\mathcal{F}$LORA-*2*: a formula is compiled into an object that represents that formula if that formula is wrapped with the ${...} construct as in

```
?- foobar(${a[b->?X]}, ?X).
```

Reification is further discussed in Section 12.2.

9 Boolean Methods

As a syntactic sugar, FLORA-2 provides boolean methods, which can be considered as value-returning methods that return some fixed value, e.g., `void`. For example, the following facts:

```
John[is_tall -> void].
John[loves(tennis) -> void].
```

can be simplified as boolean methods as follows:

```
John[is_tall].
John[loves(tennis)].
```

Conceptually, boolean methods are statements about objects whose truth value is the only concern. Boolean methods do not return any value (not even the value `void`). Therefore, boolean methods `cannot` appear in path expressions. For instance, `John.is_vegetarian`, where `is_vegetarian` is a binary method, is illegal.

Like other methods, boolean methods can be inheritable. To make a boolean method inheritable, the `*` sign is prepended to the method name:

```
buddhist[*is_vegetarian].
John:buddhist.
```

The above says that all Buddhists are vegetarian and John (the object with oid `John`) is a Buddhist. Since `is_vegetarian` is inheritable, it follows that John is also a vegetarian, i.e., `John[is_vegetarian]`.

9.1 Boolean Signatures

Boolean methods can have signatures like value-returning methods. For noninheritable Boolean methods, signatures are specified as follows:

```
Class[=>Meth]
```

For inheritable Boolean methods, signatures are declared similarly:

```
C[*=>Meth]
```

10 Anonymous and Generated Oids

For applications where oids are not important, FLORA-2 provides the compiler directive `_#` to automatically generate a new oid. `_#` can be used wherever an Id-term is allowed, *except in the*

rule body, where such oids make no sense. Like the anonymous variable `?_`, each occurrence of `_#` represents an *anonymous oid*. The difference is that such an oid is not only unique in each rule, but in the source program as well.

Of course, uniqueness is achieved through the use of special "weird" naming schema for such oids, which internally prefixes them with several ``_$''s. However, as long as the user does not use a similar naming convention (who, on earth, would give names that begin with lots of ``_$''s?), uniqueness is guaranteed.

For example, in the following program:

```
_#[ssn->123, father->_#[name->John, spouse->_#[name->Mary]]].
foo[_#(?X)->?Y] :- bar[?Y->?X].
```

the compiler will generate unique oids for each occurrence of `_#`. Note that, in the second clause, only one oid is generated and it serves as a method name.

In some situations, it is needed to be able to create a new oid and use it within the same rule head or a fact. Since such an oid needs to be referenced inside the same program clause, it is no longer possible to use `_#`, because each occurrence of `_#` causes the compiler to generate a new oid. To solve this problem, FLORA-*2* allows *numbered anonymous oids*, which are of the form `_#132`, *i.e.*, `_#` with a number attached to it. For instance,

```
_#1[ssn->123, father->f(_#1)[name->John, spouse->_#[name->Mary]]].
_#1[self->_#1].
```

The first time the compiler finds `_#1` in the first clause above, it will generate a new oid. However, the second occurrence of `_#1` in the *same* clause (*i.e.*, `f(_#1)`) will use the oid previously generated for the first occurrence. On the other hand, occurrences of `_#1` in *different* clauses are substituted with different oids. Thus, the occurrences of `_#1` in the first and second clauses above refer to different objects.

Anonymous oids are generated at compile time without regard for the oids that might exist at run time. Sometimes it is necessary to generate a completely new oid at *run time*. This can be accomplished with the `newoid{...}` builtin. For instance,

```
flora2 ?- newoid{?X}.

?X = _$_$_flora'dyn_newoid308

1 solution(s) in 0.0000 seconds

Yes
```

11 Multifile Programs

FLORA-*2* supports many ways in which a program can be modularized. First, an F-logic program

can be split into many files with separate namespaces. Each such file can be considered an independent library, and the different libraries can call each other. In particular, the same method name (or a predicate) can be used in different files and the definitions will not clash. Second, a program file can be split of several files, and these files can be included by the preprocessor prior to the compilation. In this case, all files share the same namespace in the sense that the different rules that define the same method name (or a predicate) in different files are assumed to be part of one definition. Third, $\mathcal{F}$LORA-*2* programs can call Prolog modules and vice versa. In this way, a large system can be built partly in Prolog and partly in $\mathcal{F}$LORA-*2*.

We discuss each of the aforesaid modularization methods in turn.

11.1 $\mathcal{F}$LORA-*2* Modules

A $\mathcal{F}$LORA-*2* *module* is a programming abstraction that allows a large program to be split into separate libraries that can be reused in multiple ways in the same program. Formally, a module is a pair that consists of a *name* and a *contents*. The name must be an alphanumeric symbol (the underscore, _, is also allowed), and the contents consists of the program code that is typically loaded from some file (but it can also be constructed dynamically by inserting facts into another module).

The basic idea behind $\mathcal{F}$LORA-*2* modularization is that reusable code libraries are to be placed in separate files. To use a library, it must be *loaded into a module*. Other parts of the program can then invoke this library's methods by providing the name of the module (and the method/predicate names, of course). There is no need to export anything from a library — any public method or predicate can be called by other parts of the program. (A module can have non-public methods, if the module is encapsulated — see Section 11.12.) In this way, the library loaded into a module becomes that module's content.

Note that there is no a priori association between files and modules. Any file can be loaded into any module and one program file can even be loaded into two different modules at the same time. The same module can be reused during the same program run by loading another file into that module. In this case, the old contents is erased and the module gets new contents from the second file.

In $\mathcal{F}$LORA-*2*, modules are completely decoupled from file names. A $\mathcal{F}$LORA-*2* program knows only the module names it needs to call, but not the file names. Specific files can be loaded into modules by another, unrelated bootstrapping program. Moreover, a program can be written in such a way that it calls a method of some module without knowing that module's name. The name of the module can be passed as a parameter or in some other way and the concrete binding of the method to the module will be done at runtime.

This dynamic nature of $\mathcal{F}$LORA-*2* modules stands in sharp contrast to the module system of Prolog, which is static and associates modules with files at compile time. Moreover, to call a predicate from another module, that predicate must be imported explicitly and referred to by the same name.

As a pragmatic measure, $\mathcal{F}$LORA-*2* defines *three kinds of modules* rather than just one. The kind described above is actually just one of the three: the *user module*. As explained, these modules

are decoupled from the actual code, and so they can contain different code at different times. The next kind is a *Prolog module*. This is an abstraction in FLORA-*2*, which is used to call Prolog predicates. Prolog modules are static and are assumed to be closely associated with their code. We describe these modules in Section 11.7. (Do not confuse FLORA-*2* Prolog modules — an abstraction used in the language of FLORA-*2*— with Prolog modules, which is an abstraction used in Prolog.) The third type of modules are the FLORA-*2* `system modules`. These modules are preloaded with FLORA-*2* programs that provide useful methods and predicates (*e.g.*, I/O) and, thus, are also static. These modules are described in Section 11.9 and 29. The abstraction of system modules is a convenience provided by FLORA-*2*, which enables user programs to perform common actions using standard names of predicates and methods implemented in those modules. The syntactic conventions for calling each of these types of modules are similar, but distinct.

11.2 Calling Methods and Predicates Defined in User Modules

If *literal* is an F-molecule or a predicate defined in another user module, it can be called using the following syntax:

literal @ *module*

The name of the module can be any alphanumeric symbol.[6] For instance, `foo(a) @ foomod` tests whether `foo(a)` is true in the user module named `foomod`, and `Mary[children -> ?X]@genealogy` queries the information on `Mary`'s children available in the module `genealogy`. More interestingly, the module specifier can be a variable that gets bound to a module name at run time. For instance,

```
..., ?Agent=zagat, ..., newyork[dinner(italian) -> ?X]@?Agent.
```

A call to a literal with an unbound module specification or one that is not bound to a symbol will result in a runtime error.

When calling the literals defined in the same module, the `@`*module* notation is not needed, of course. (In fact, since a program does not know where it will be loaded, using the @-notation to call a literal in the same module is hard. However, it is possible with the help of the special token `_@`, which is described later, and is left as an exercise.)

The following rules apply when calling a literal defined in another module:

1. Literal reference cannot appear in a rule head or be specified as a fact. For example, the following program will generate a parsing error

   ```
   John[father->Smith] @ foomod.
   foo(?X) @ foomod :- goo(?X).
   ```

   because defining a literal that belongs to another module does not make sense.

[6] In fact, any symbol is allowed. However, it cannot contain the quote symbol, "'".

2. Module specification is distributive over logical connectives, including the conjunction operator, "`,`", the disjunction, "`;`", and the negation operators, "`\+`" and "`not`". For example, the formula below:

   ```
   (John[father->Smith], not Smith[spouse->Mary]) @ foomod
   ```

   is equivalent to the following formula:

   ```
   John[father->Smith] @ foomod, not (Smith[spouse->Mary] @ foomod)
   ```

3. Module specifications can be nested. The one closest to a literal takes effect. For example,

   ```
   (foo(a), goo(b) @ goomod, hoo(c)) @ foomod
   ```

   is equivalent to

   ```
   foo(a) @ foomod, goo(b) @ goomod, hoo(c) @ foomod
   ```

4. The module specification propagates to any F-molecule appearing in the argument of a predicate for which the module is specified. For example,

   ```
   foo(a.b[c->d]) @ foomod
   ```

   is equivalent to

   ```
   a[b->?X] @ foomod, ?X[c->d] @ foomod, foo(?X) @ foomod
   ```

5. Module specifications do not affect function terms that are not predicates or method names, unless such a specification is explicitly attached to such a term. For instance, in

   ```
   flora2 ?- foo(goo(a)) @ foomod.
   ```

   `goo/1` refers to the same functor both in module `foomod` and in the calling module. However, if the argument is *reified* (i.e., is an object that represents a formula — see Section 12.2), as in

   ```
   flora2 ?- foo(${goo(a) @ goomod}) @ foomod.
   ```

   then `foo/1` is assumed to be a meta-predicate that receives the query `goo(a)` in module `goomod` as a parameter. Moreover, module specification propagates to any reified formula appearing in the argument of a predicate for which the module is specified. For example,

   ```
   flora2 ?- foo(${goo(a)}) @ foomod.
   ```

   is equivalent to

   ```
   flora2 ?- foo(${goo(a) @ foomod}) @ foomod.
   ```

11.3 Finding the Current Module Name

Since a FLORA-2 program can be loaded into any module, the program does not have a priori knowledge of the module it will be executing in. However, the program can determine its module at runtime using the special token `_@`, which is replaced with the current module name when the module is loaded. More precisely, if `_@` occurs anywhere as an oid, method name, value, etc., in file `foo.flr` then when `foo.flr` is loaded into a module, say, `bar`, then all such occurrences of `_@` are replaced with `bar`. For instance,

```
a[b->_@].
?- a[b->?X].

?X=main

Yes
```

11.4 Finding the Module That Invoked A Rule

Sometimes it is useful to find out which module called any particular rule at run time. This can be used, for example, when the rule performs different services for different modules. The name of the caller-module can be obtained by calling the primitive `caller{?X}` in the body of a rule. For instance,

```
p(?X) :- caller{?X}, (write('I was called by module: '), writeln(?X))@_prolog.
```

When a call to predicate `p(?X)` is made from any module, say `foobar`, and the above rule is invoked as a result, then the message "I was called by module: foobar" will be printed.

11.5 Loading Files into User Modules

FLORA-2 provides several commands for compiling and loading program files into specified user modules.

Compilation. The command

`flora2 ?- _compile(`*file*`>>`*module*`).`

generates the byte code for the program to be loaded into the user module named `module`. The name of the byte code for the program in *file*.flr, which can later be loaded into the specified module. In practice this means that the compiler generates files named *file_module.P* and *file_module.xwam* with symbols appropriately renamed to avoid clashes.

If no module is specified, the command

`flora2 ?- _compile(`*file*`).`

compiles *file*.flr for the default module `main`.

Loading. The above commands compile files without actually loading their contents into the in-memory knowledge base. To load a file, the following commands can be used:

```
flora2 ?- [myprog].
flora2 ?- _load(myprog).
```

This loads the program in the file `myprog.flr` into the default user module `main`. If `myprog.flr` is newer than the compiled code, the source file is recompiled.

An optional module name can be given to tell FLORA-*2* to load the program into the specified module:

```
flora2 ?- [myprog >> foomod].
flora2 ?- _load(myprog >> foomod).
```

This loads the FLORA-*2* program `myprog.flr` into the user module named `foomod`, compiling it if necessary.

The user can compile and load several program files at the same time: If the file was not compiled before (or if the program file is newer), the program is compiled before being loaded. For instance, the following command:

```
flora2 ?- [myprog1, myprog2]
```

will load both `myprog1` and `myprog2` into the default module `main`. However, loading several programs into the same module is not very useful: the code of the last program will wipe out the code of the previous ones. This is a general rule in FLORA-*2*. Thus, loading multiple files is normally used in conjunction with the module targets:

```
flora2 ?- ['myprog1.flr', myprog2 >> foomod].
```

which loads `myprog1.flr` into the module `main` and `myprog2.flr` into the module `foomod`.

Adding to already loaded modules. Files can also be *added* to an existing module, as explained in the following subsection.

Note that the `[...]` command can also load and compile Prolog programs. The overall algorithm is as follows. If the file suffix is specified explicitly, the corresponding file is assumed to be a FLORA-*2* file, a Prolog file, or a byte code depending on the suffix: `.flr`, `.P`, or `.xwam`. If the suffix is not given explicitly, the compiler first checks if *file*.`flr` exists. If so, the file assumed to be a FLORA-*2* program and is compiled as such. If *file*.`flr` is not found, but *file*.`P` or *file*.`O` is, the file is passed to Prolog for compilation.

Sometimes it is useful to know which user modules are loaded or if a particular user module is loaded (say, because your might want to load it, if not). To find out which modules are loaded at

the present time, use the predicate `_isloaded/1`. For instance, the first query, below, succeeds if the module `foo` is loaded. The second query succeeds and binds `L` to the list of all user modules that are loaded at the present time (aggregate operators, including `collectset` are discussed in Section 21.

```
flora2 ?- _isloaded(foo).
flora2 ?- ?L= collectset{?X|_isloaded(?X)}.
```

Inline programs. In some cases—primarily for testing—it is convenient to be able to type up and load small programs into a running FLORA-*2* session. To this end, the system provides special idioms, `[_]`, `[_>>module]`, `[+_]`, and `[+_>>module]`. This causes FLORA-*2* to start reading program clauses from the standard input and load them into the default module or the specified module. To indicate the end of the input, the user can type `Control-D` in Unix-like systems or `Control-Z` in Windows. For instance,

```
flora2 ?- [_>>foo].
aaa[bbb->ccc].
?X[foo->?Y] :- ?Y[?X->bar].
Control-D
```

A word of caution. It is dangerous to place the `_load` command in the body of a rule if `_load` loads a file into the same module where the rule belongs. For instance, if the following rule is in module `bar`

```
p(X) :- ..., [foo>>bar], ...
```

then execution of such a rule is likely to crash Prolog. This is because this very rule will be wiped out before it finishes execution — something that XSB is not ready for. FLORA-*2* tries to forewarn the .user about such dangerous occurrences of `_load`, but it cannot intercept all such cases reliably.

11.6 Adding Rule Bases to Existing Modules

Loading a file into a module causes the knowledge base contained in that module to be erased before the new information is loaded. Sometimes, however, it is desirable to *add* knowledge (rules and facts) contained in a file to an existing module. This operation does not erase the old contents of the module. For instance each of the following commands

```
flora2 ?- [+myprog >> foomod].
flora2 ?- _add(myprog >> foomod).
```

will add the rules and facts contained in t=file `myprog.flr` into the module `foomod` without erasing the old contents. The following commands

```
flora2 ?- [myprog].
flora2 ?- _load(myprog).
```

will do the same for module `main`. Note that, in the `[...]` form, loading and adding can be freely mixed. For instance,

```
flora2 ?- [foo1, +foo2]
```

will first load the file `foo1.flr` into the default module `main` and then add the contents of `foo2.flr` to that same module.

Like the loading commands, the addition statements first compile the files they load if necessary. It is also possible to compile files for *later* addition without actually adding them. Since files are compiled for addition a little differently from files compiled for loading, we use a different command:

```
flora2 ?- _compileadd(foo).
flora2 ?- _compileadd(foo >> bar).
```

11.7 Calling Prolog from *FLORA-2*

Prolog predicates can be called from *FLORA-2* through the *FLORA-2* module system *FLORA-2* models Prolog programs as collections of static *Prolog modules*, *i.e.*, from *FLORA-2*'s point of view, Prolog modules are always available and do not need to be loaded explicitly because the association between Prolog programs and modules is fixed.

@_prolog and @_plg. The syntax to call Prolog predicates is one of the following:

```
flora2 ?- predicate@_prolog(module)
```

For instance, since the predicate `member/2` is defined in the Prolog module `basics`, we can call it as follows:

```
flora2 ?- member(abc,[cde,abc,pqr])@_prolog(basics).
```

`_plg` instead of `_prolog` also works.

To use this mechanism, you must know which Prolog module the particular predicate is defined in. Some predicates are defined by programs that do not belong to any module. When such an Prolog program is loaded, the corresponding predicates become available in the default Prolog module. In XSB, the default module is called `usermod` and *FLORA-2* can call such predicates as follows:

```
flora2 ?- foo(?X)@_prolog(usermod).
```

Note that variables are not allowed in the module specifications of Prolog predicates, *i.e.*,

```
flora2 ?- ?M=usermod, foo(?X)@_prolog(?M).
```

will cause a compilation error.

Some Prolog predicates are considered "well-known" and, even though they are defined in various Prolog modules, the user can just use those predicates without remembering the corresponding Prolog module names. These predicates (that are listed in the XSB manual) can be called from FLORA-2 with particular ease:

```
flora2 ?- writeln('Hello')@_prolog
```

i.e., we can simply omit the Prolog module name (but parentheses must be preserved).

@_prologall and @_plgall. The Prolog module specification `@_prolog` has one subtlety: it does not affect the arguments of a call. For instance,

```
flora2 ?- foo(f(?X,b))@_prolog.
```

will call the Prolog predicate foo/1. Recall that FLORA-2 uses HiLog terms to represent objects, while Prolog uses Prolog terms. Thus, the argument `f(?X,b)` above will be treated as a HiLog term. Although it looks like a Prolog term and, in fact, HiLog terms generalize Prolog terms, the internal representation of HiLog and Prolog terms is different. Therefore, if the fact `foo(f(a,b))` is defined somewhere in the Prolog program then the above query will fail, since a Prolog term `f(?X,b)` and a HiLog term `f(?X,b)` are *different* even though their textual representation in FLORA-2 is the same.

A correct call to `foo/1` in this case would be as follows:

```
?- foo(f(?X,b)@_prolog)@_prolog.
```

Here we explicitly tell the system to treat `f(?X,b)` as a Prolog term. Clearly, this might be too much writing in some cases, and it is also error prone. Moreover, bindings returned by Prolog predicates are Prolog terms and they somehow need to be converted into HiLog.

To simplify calls to Prolog, FLORA-2 provides another, more powerful primitive: `@_prologall`. In the above case, one can call

```
?- foo(f(?X,b))@_prologall.
```

without having to worry about the differences between the HiLog representation of terms in FLORA-2 and the representation used in Prolog.

One might wonder why is there the `@_prolog` module call in the first place. The reason is efficiency. The `@_prologall` call does automatic conversion between Prolog and HiLog, which is not always necessary. For instance, to check whether a term, `f(a)`, is a member of a list, `[f(b),f(a)]`, one does not need to do any conversion, because the answer is the same whether these terms are HiLog terms or Prolog terms. Thus,

```
?- member(f(a), [f(b),f(a)])@_prolog(basics).
```

is perfectly acceptable and is more efficient than

```
?- member(f(a), [f(b),f(a)])@_prologall basics.
```

$\mathcal{F}$LORA-*2* provides a special primitive, `p2h{...,...}`, which converts terms to and from the HiLog representation, and the programmer can use it in conjunction with `@_prolog` to achieve a greater degree of control over argument conversion. This issue is further discussed in Section 12.4.

11.8 Calling $\mathcal{F}$LORA-*2* from Prolog

Since Prolog does not understand $\mathcal{F}$LORA-*2* syntax, it can call only predicates defined in $\mathcal{F}$LORA-*2* programs. To call predicates defined in $\mathcal{F}$LORA-*2* programs, they must be imported by the Prolog program.

11.8.1 Importing $\mathcal{F}$LORA-*2* Predicates into Prolog Shell

To import a $\mathcal{F}$LORA-*2* predicate into Prolog shell, the following must be done:

- The query

  ```
  | ?- [flora2], bootstrap_flora.
  ```

  must be executed first.

- One of the following `'_flimport'` queries must be executed in the shell:

  ```
  | ?- '_flimport' flora-predicate/arity as xsb-name(_,_,...,_)
                                from filename >> flora-module-name
  | ?- '_flimport' flora-predicate/arity as xsb-name(_,_,...,_)
                                from flora-module-name
  ```

We will explain shortly which `'_flimport'` query should be used in what situation.

Note: If $\mathcal{F}$LORA-*2* is installed outside of the XSB directory structure, then you must let Prolog know the location of your installation of $\mathcal{F}$LORA-*2*. This is done by executing the prolog instruction `asserta(library_directory(path-to-flora))`. For instance

```
?- asserta(library_directory('/home/me/flora2')).
```

before calling any of the $\mathcal{F}$LORA-*2* modules. Observe that `asserta` and *not* `assert` must be used.

11.8.2 Calling FLORA-2 from a Prolog Module

To call FLORA-2 from within a Prolog program, say `test.P`, the following must be done:

1. The query

   ```
   ?- [flora2], bootstrap_flora
   ```

   must be executed *before compiling or loading* `test.P` — otherwise, the program will not compile or load.

2. The directive

   ```
   :- import ('_flimport')/1 from flora2.
   ```

   must appear near the top of `test.P`, *prior to any call* to FLORA-2 predicates.

The first form for `'_flimport'` above is used to both import the predicate and also to load the program file defining it into a given FLORA-2 user module. The second syntax is used when the FLORA-2 program is already loaded into a module and we only need to import the corresponding predicate.

In `'_flimport'`, *flora-predicate* is the name of the imported predicate as it is known in the FLORA-2 module. For non-tabled predicates, whose name starts with `%` in FLORA-2, *flora-predicate* should have the following syntax: `%(predicate-name)`. For instance, to import a FLORA-2 non-tabled predicate `%foobar` of arity 3 one can use the following statement:

```
?- '_flimport' '%'(foobar)/3 as foobar(_,_,_) from mymodule.
```

The imported predicate must be given a name by which the imported predicate will be known in Prolog. (This name can be the same as the name used in FLORA-2.) It is important, however, that the Prolog name be specified as shown, *i.e.*, as a predicate skeleton with the same number of arguments as in the FLORA-2 predicate. For instance, `foo(_,_,_)` will do, but `foo/3` will not. Once the predicate is imported, it can be used under its Prolog name as a regular predicate.

Prolog programs can also load and compile FLORA-2 programs using the following queries (again, `bootstrap_flora` must be executed in advance):

```
:- import '_load'/1, '_compile'/1 from flora2.
?- '_load'(flora-file >> flora-module).
?- '_load'(flora-file).
?- '_compile'(flora-file >> flora-module).
?- '_compile'(flora-file).
```

The first query loads the file *flora-file* into the given user module and compiles it, if necessary. The second query loads the program into the default module `main`. The last two queries compile the file for loading into the module *flora-module* and `main`, respectively, but do not load it.

Finally, a Prolog program can check if a certain $\mathcal{F}$LORA-*2* user module has been loaded using the following call:

```
:- import '_isloaded'/1 from flora2.
?- '_isloaded'(flora-module-name).
```

Note that when used inside Prolog the `_isloaded` predicate must be quoted. Quoting is not necessary when it is used in $\mathcal{F}$LORA-*2*.

Note: If you are using a release of $\mathcal{F}$LORA-*2* that is installed outside of the XSB directory tree, you must make sure that Prolog will find this installation and use it. One way of doing this was described earlier (by executing an appropriate `asserta/1`). This method works best if your application consists of both $\mathcal{F}$LORA-*2* and Prolog modules, but the initial module of your application (*i.e.*, the one that bootstraps everything) is a Prolog program. If the initial module is a $\mathcal{F}$LORA-*2* program, then the best way is to start XSB and $\mathcal{F}$LORA-*2* using the `runflora` script (page 3) located in the distribution of $\mathcal{F}$LORA-*2*.

11.8.3 Passing Arbitrary Queries to $\mathcal{F}$LORA-*2*

The method of calling $\mathcal{F}$LORA-*2* from Prolog, which we just described, assumes that the user knows which predicates and methods to call in the $\mathcal{F}$LORA-*2* module. Sometimes, it is useful to be able to pass arbitrary queries to $\mathcal{F}$LORA-*2*. This is particularly useful when $\mathcal{F}$LORA-*2* runs under the control of a Java or C program.

To enable such unrestricted queries, $\mathcal{F}$LORA-*2* provides a special predicate, `flora_query/4`, which is called from Prolog and takes the following arguments:

- *String*: A string that contains a $\mathcal{F}$LORA-*2* query. It can be an atom (e.g., 'foo[bar-¿?X].') or a list of character codes (e.g., "foo[bar-¿?X].").
- *Vars*: A list of the form `['?Name1'=Var1, '?Name2'=Var2,...]` or of the form `["?Name1"=Var1, "?Name2"=Var2,...]`. `?Name` is a name of a variable mentioned in *String*, for instance, `'?X'` (note: the name must be quoted, since it is an atom). Var is a *Prolog* (not $\mathcal{F}$LORA-*2*!) variable where you want the binding for the variable `Name` in *String* to be returned. For instance, if *String* is `'p(?X,?Y).'` then *Vars* can be `['?X' = Xyz, "?Y" = Qpr]`. In this case, `Xyz` will be bound to the value of `?X` in `p(?X,?Y)` after the execution, and `Qpr` will be bound to the value of `?Y` in `p(?X,?Y)`.
- *Status*: Indicates the status of compilation of the command in `String`. It is a list, which contains various indicators. The most important ones are `success` and `failure`.
- *Exception*: If the execution of the query is successful, this variable is bound to `normal`. Otherwise, this variable will contain an exception term returned by XSB (see the XSB manual, if you need to process exceptions in sophisticated ways).

In order to use the the `flora_query/4` predicate from within Prolog, the following steps are necessary.

1. If $\mathcal{F}$LORA-*2* is installed as a standalone application rather than an XSB package, then the $\mathcal{F}$LORA-*2* installation directory must be added to the XSB search path:

   ```
   ?- asserta(library_directory('/home/myHomeDir/flora2')).
   ```

2. The query

   ```
   ?- [flora2], bootstrap_flora
   ```

   must be executed *before compiling or loading* the Prolog file.

3. `flora_query/4` must be imported from `flora2`.

Here is an example of a Prolog file, `test.P`, which loads and then queries a $\mathcal{F}$LORA-*2* file, flrtest.flr:

```
:- import bootstrap_flora/0 from flora2.
?- asserta(library_directory('/home/myHomeDir/flora2')),
   [flora2],
   bootstrap_flora.
:- import flora_query/4 from flora2.
:- import '_load'/1 from flora2.

?- '_load'(flrtest).

?- Str="?X[b->?Y].", flora_query(Str,["?X"=YYY,"?Y"=PPP], _Status,_Exception).
```

After the query to `flrtest.flr` is successfully executed, the bindings for the variable `?X` in the $\mathcal{F}$LORA-*2* query will be returned in the Prolog variable `YYY`. The binding for `?Y` in the query will be returned in the Prolog variable `PPP`. If there are several answers, you can get them all in a failure-loop, as usual in Prolog. For instance,

```
?- Str='?X[b->?Y].', flora_query(Str,['?X'=YYY,'?Y'=PPP], _Status,_Exception),
   writeln('?X' = YYY),
   writeln('?Y' = PPP),
   fail.
```

Note that the Prolog variables in the variable list (like `YYY` and `PPP` above) can be bound and in this way input to the $\mathcal{F}$LORA-*2* query can be provided. For instance,

```
?- YYY=abc, flora_query('?X[b->?Y].',['?X'=YYY,'?Y'=PPP], _Status,_Exception).
```

yields the same result as

```
?- flora_query('abc[b->?Y].',['?Y'=PPP], _Status,_Exception).
```

However, the user should be aware of the fact that if a query is going to be used many times with different parameters then the first form is much faster. That is,

```
?- YYY=abc1, flora_query('?X[b->?Y].',['?X'=YYY,'?Y'=PPP], _Status,_Exception).
?- YYY=abc2, flora_query('?X[b->?Y].',['?X'=YYY,'?Y'=PPP], _Status,_Exception).
......
```

is noticeably faster than

```
?- flora_query('abc1[b->?Y].',['?Y'=PPP], _Status,_Exception).
?- flora_query('abc2[b->?Y].',['?Y'=PPP], _Status,_Exception).
......
```

if the above queries are executed thousands of times with different parameters `abc1`, `abc2`, etc.

11.9 $\mathcal{F}$LORA-*2* System Modules

$\mathcal{F}$LORA-*2* provides a special set of modules that are *preloaded* with useful utilities, such as prettyprinting or I/O. These modules have special syntax, `_modname`, and cannot be loaded by the user. For this reason, these modules are called $\mathcal{F}$LORA-*2 system modules*. For instance, to prettyprint all the attributes and methods of an object, the following method, defined in the system module `pp`, can be used:

```
flora2 ?- obj[%pp_self]@_pp.
```

Here, the method `%pp_self` is applied to the object `obj` and will pretty-print the state of that object. For more details on the existing $\mathcal{F}$LORA-*2* system modules, see Section 29.

11.10 Including Files into $\mathcal{F}$LORA-*2* Programs

The last and the simplest way to construct multi-file $\mathcal{F}$LORA-*2* programs is by using the `#include` preprocessing directive. For instance if file `foo.flr` contains the following instructions:

```
#include "file1"
#include "file2"
#include "file3"
```

the effect is the same as if the above three files were concatenated together and stored in `foo.flr`. Note, however, that when compiling `foo.flr`, the compiler has no way of knowing if any of the included files have changed, because file inclusion is done by the preprocessor. So, it is recommended to compile such multi-file programs using a Makefile, like in C and C++.

Note that the `#include` instruction requires that the file name is enclosed in *double* quotes. Also, under Windows, backslashes in file names must be doubled. For instance,

```
#include "..\\foo\\bar.flr"
```

11.11 More on Variables as Module Specifications

Earlier we mentioned that a user module specification can be a variable, *e.g.*, `a[m->b]@?X`, which ranges over module names. This variable does not need to be bound to a concrete module name before the call is made. If it is a variable, then `?X` will get successively bound to the user modules where `a[m->b]` is true. However, these bindings will not include `_prolog()`, `_prolog(module)`, or `_module`.

Dynamic module bindings can be used to implement *adaptive methods*, which are used in many types of applications, *e.g.*, agent programming. Consider the following example:

```
Module foo                                      Module moo
something :- ...                                ......
something_else :-                               ......
        a[someservice(_@,?Arg)->?Res]@moo       ......
......                                          a[someservice(?Module,?Arg)-> ?Res] :-
......                                                      something@?Module, ...
......                                          ......
```

Here the method `someservice` in user module `moo` performs different operation depending on who is calling it, because `something` can be defined differently for different callers. When `something_else` is called in module `foo`, it invokes the method `someservice` on object `a` in module `moo`. The current module name (`foo`) is passed as a parameter (with the token `_@`). When `someservice` is executed in module `moo` it therefore calls the predicate `something` in module `foo`. If `someservice` is called from a different module, say `bar`, it will invoke `something` defined in *that* module and the result might be different, since `something` in module `bar` may have a different definition than in module `foo`.

An example of the use of the above idea is the pretty printing module of $\mathcal{F}$LORA-*2*. A pretty-printing method is called on an object in some user module, and to do its job the pretty-printing method needs to query the object *in the context of the calling module* to find the methods that the object has.

It is also possible to view adaptive methods as a declarative counterpart of the callback functions in C/C++, which allows the callee to behave differently for different clients.

11.12 Module Encapsulation

So far in multi-module programs any module could call any method or predicate in any other module. That is, modules were not encapsulated. However, $\mathcal{F}$LORA-*2* lets the user to encapsulate any module and export the methods and predicates that other modules are allowed to call. Making an unexported call will result in a runtime error.

A module is encapsulated by placing an `export` directive in it or by executing an `export` directive at run time. Modules that do not have `export` directives in them are not encapsulated, which means that any method or predicate defined inside such a module can be called from the outside.

Syntax. The `export` directive has the form:

`:- export` *MethodOrPredExportSpec1*, *MethodOrPredExportSpec2*, ...

There can be one or more export specifications (*MethodOrPredExportSpec*) in each `export` statement, and there can be any number of different `export` statements in a module. The effect of all these statements is cumulative.

Each *MethodOrPredExportSpec* specifies three things, two of which are optional:

- The list of methods or predicates to export.
- The list of modules to which to export. This list is optional. If it is not given then the predicates and modules are exported to *all* modules.
- Whether the above are exported as *updatable* or not. If a method or a predicate is exported as updatable, then the external modules can add or delete the corresponding facts. Otherwise, these modules can only query these methods and predicates. If `updatable` is not specified, the calls are exported for querying only.

The exact syntax of a *MethodOrPredExportSpec* is as follows:

[`updatable`] *ExportList* [`>>` *ModuleList*]

The square brackets here denote optional parts. The module list is simply a comma-separated list of modules and *ExportList* is a comma-separated list of *predicate/method/ISA templates*. Method templates have the form

`?[` *termTemplate* `-> ?]` or
`?[` *termTemplate* `]`

and predicate templates are the same as term templates. A *term template* is a HiLog term that has no constants or function symbols in it. For instance, `p(?,?)(?)` and `q(?,?,?)` are term templates, while `p(a,?)(?)` and `q(?,?,f(?))` are not.

ISA templates have the form `?:?` or `?::?`. Of course, `?_` can also be used instead of `?`.

Examples. Here are some examples of export directives:

```
:- export ?[a(?) -> ?].
:- export ?[b ->?], ?[c(?,?)], ?[d(?)(?,?) -> ?].
:- export (?[e -> ?], ?[f(?,?)]) >> (foo, bar).
:- export updatable (?[g -> ?], ?[h(?,?)]) >> (foo, bar).
:- export updatable (?[g -> ?], ?[h(?,?)]) >> (foo, bar),
            (?[k -> ?], m(?,?)(?)) >> abc.
```

Observe that the method `g` and the boolean method `h` have been exported in the updatable mode. This means that the modules `foo` and `bar` can insert and delete the facts of the form `a[g->b]` and `a[h(b,c)]` using the statements like (assuming that `moo` is the name of the module that includes the above directives):

```
?- insert{a[g->b]@moo}.
?- delete{a[h(b,c)]@moo}.
```

Parenthesizing rules. Note that in the last three export statements above we used parentheses to disambiguate the syntax. Without the parentheses, these statements would be understood differently:

```
:- export ?[e -> ?], (?[f(?,?)] >> foo), bar.
:- export updatable ?[g -> ?], (?[h(?,?)]) >> foo), bar.
:- export updatable ?[g -> ?], (?[h(?,?)] >> foo), bar,
            ?[k -> ?], (m(?,?)(?) >> abc).
```

We should also note that `updatable` binds stronger than the comma or `>>`, which means that an `export` statement such as the one bellow

```
:- export updatable ?[g -> ?], ?[h(?,?)] >> foo.
```

is actually interpreted as

```
:- export updatable(?[g -> ?]), (?[h(?,?)]) >> foo).
```

Exporting molecules other than `->`. In order to export any kind of call to a non-Boolean method, one should use only `->`. This will allow other modules to make calls, such as `a[d(c)(e,f) *-> ?X]`, `a[b ->-> ?Z]`, and `c[e=>t]` to the exported methods. The `export` directive does not allow the user to separately control calls to the molecules that involve the method specifiers such as `->*`, `=>`, `->->`, etc.

The `export` directive has an executable counterpart. For instance, at run time a module can execute an export instruction such as

```
?- export ?[e -> ?], (?[f(?,?)] >> foo), bar.
```

and export the corresponding methods. If the module was not encapsulated before, it will become now. Likewise, it is possible to execute export directives in another module. For instance executing

```
?- (export ?[e -> ?], (?[f(?,?)] >> foo), bar)@foo.
```

will cause the module `foo` to export the specified methods and to encapsulate it, if it was not encapsulated before.

11.13 Importing Modules

Referring to methods and predicates defined in other modules is one way to invoke knowledge defined separately in another program. Sometimes, however, it is convenient to *import* the entire module into another module. This practice is particularly common when it comes to reusing ontologies.

$\mathcal{F}$LORA-*2* supports import of modules through the `importmodule` compile-time directive. Its syntax is as follows:

```
:- importmodule  module1,  module2, ...,  module-k.
```

Once a module is imported, its methods and predicates can be referenced without the need to use the *module* idiom.

Importing a module is *not* the same as including another module as a file with the `#include` statement. First, only *exported* methods and predicates can be referenced by the importing module. The non-exported elements of an imported module are encapsulated. Second, even when everything is exported (as in the case when no explicit `export` directive is provided), import is still different from inclusion. To see why, consider one module, `main`, that looks like this:

```
?- [myprog>>foo].
:- importmodule foo.

p(abc).
?- q(?X).
```

This module loads a program from the file `myprog.flr` into a module `foo` and then imports that module. The importing module itself contains a fact and a query.

Suppose `myprog.flr` is as follows:

```
q(?X) :- p(?X).
p(123).
```

It is easy to see that the query `q(?X)` in the importing module `main` will return the answer `?X = 123`. In contrast, if the module `main` *included* `myprog.flr` instead of importing it, i.e., if it looked like this:

```
#include "myprog.flr"
p(abc).
?- q(?X).
```

then the same query would have returned two answers: `?X = 123` and `?X = abc`. This is because the latter program is simply

```
q(?X) :- p(?X).
p(123).
p(abc).
?- q(?X).
```

In other words, in the first case, the query `q(?X)` still queries module `foo` even though the query does not use the *foo* idiom. The module `foo` has only one answer to the query, so only one answer is returned. In contrast, when `myprog.flr` is included then the resulting program has two `p`-facts and two answers are returned.

11.14 Persistent Modules

Normally, the data in a $\mathcal{F}$LORA-*2* module is *transient* — it is lost as soon as the system terminates. The $\mathcal{F}$LORA-*2* package `persistentmodules` allows one to make $\mathcal{F}$LORA-*2* modules *persistent*. This package is described in the document *A Guide to* $\mathcal{F}$LORA-*2 Packages*.

12 HiLog and Metaprogramming

HiLog [4] is the default syntax that $\mathcal{F}$LORA-*2* uses to represent functor terms (including object Ids) and predicates. In HiLog, complex terms can appear wherever a function symbol is allowed. For example, `group(?X)(?Y,?Z)` is a HiLog term where the functor is no longer a symbol but rather a complex term `group(?X)`. Variables in HiLog can range over terms, predicate and function symbols, and even over atomic formulas. For instance,

$$? - \mathtt{p}(?\mathtt{X}), ?\mathtt{X}(\mathtt{p}).$$

and

$$? - \mathtt{p}(?\mathtt{X}),\ ?\mathtt{X}(\mathtt{p}),\ ?\mathtt{X}. \tag{3}$$

are perfectly legal queries. If `p(a(b))`, `a(b)(p)`, and `a(b)` are all true in the database, then $?\mathtt{X} = \mathtt{a}(\mathtt{b})$ is one of the answers to the query in HiLog.

Although HiLog has a higher order syntax, its semantics is first order [4]. Any HiLog term can be consistently translated into a Prolog term. For instance, `group(?X)(?Y,?Z)` can be represented by the Prolog term `apply(apply(group,?X),?Y,?Z)`. The translation scheme is pretty straightforward and is described in [4].

Any Id-term in $\mathcal{F}$LORA-*2*, including function symbols and predicate symbols, are considered to be HiLog terms and therefore are subject to translation. That is, even a normal Prolog term will by default be represented using the HiLog translation, e.g., `foo(a)` will be represented as `apply(foo,a)`. This guarantees that HiLog unification will work correctly at runtime. For instance, `foo(a)` will unify with `?F(a)` and bind the variable `?F` to `foo`.

There is one important difference between HiLog, as described in [4], and its implementation in $\mathcal{F}$LORA-*2*. In HiLog, functor terms that appear as arguments to predicates and the atomic formulas (*i.e.*, predicates that are applied to some arguments) belong to the same domain. In contrast, in

*F*LORA-*2* they are in different domains.[7] For instance, suppose `p(a(b))` is true, and consider the following query:

```
?- ?X ∼ a(b), p(?X).
```

Here ∼ is a meta-unification operator to be discussed shortly, in Section 12.1; it binds `?X` to the atomic formula `a(b)` in the current module. The answer to this query is 'No' because `?X` is bound to the atomic formula `a(b)`, while `a(b)` in `p(a(b))` is a HiLog term.

Our earlier query, (3), will also not work (unlike in the original HiLog) because `?X` is bound to a term and not a formula: if we execute the query (3), we will get an error stating that `?X` is bound to a HiLog term, not a predicate, and therefore the query `?X` is meaningless. To correct the problem, `?X` must be promoted to a predicate and relativized to a concrete module—in our case to the current module. So, the following query *will* work and produce a binding `a(b)` for `?X`.

```
flora2 ?- p(?X), ?X(p), ?X@ _@.
```

Like in classical logic, `foo` and `foo()` are different terms. However, in programming, it is convenient to identify these terms when they are treated as predicates. Prologs often disallow the use of the `foo()` syntax altogether. The same distinction holds in HiLog: `foo`, `foo()` and `foo()()` are all different. In terms of the HiLog to Prolog translation, this means that `foo` is different from `apply(foo)` is different from `apply(apply(foo))`. However, just like in Prolog, we treat `p` as syntactic sugar for `p()` when both occur as predicates. Thus, the following queries are the same:

```
flora2 ?- p.
flora2 ?- p().
```

In the following program,

```
p.
q().
?- p(), ?X().
?- q, ?X().
?- r = r().
```

the first two queries will succeed (with `?X` bound to `p` or `q`), but the last one will fail. Identification of `p` with `p()` does not extend to `p()()`, which is distinct from both `p` and `p()` not only as a term but also as a formula. Thus, in the following program, all queries fail:

```
p.
q().
?- p()().
?- q()().
?- p = p()().
?- q() = q()().
```

[7] This is allowed in *sorted HiLog* [3].

12.1 Meta-programming, Meta-unification

F-logic together with HiLog is powerful stuff. In particular, it 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 simple examples:

```
// all unary methods defined for John
?- John[?M(?) -> ?].

// all unary methods that apply to John,
// for which a signature was declared
?- John[?M(?) => ?].

// all method signatures that apply to John,
// which are either declared explicitly or inherited
?- John[?M => ?].

// all method invocations defined for John
?- 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 Prolog system and $\mathcal{F}$LORA-*2* simply takes advantage of them:

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

?- compound(f(?X))@_prolog.
?X = _h472
Yes
```

Note that these primitives are used for Prolog terms only and are described in the XSB manual. These primitives have not been ported to work with HiLog terms yet.

Meta-unification. In $\mathcal{F}$LORA-*2*, variables can be bound to both formulas and terms. For instance, in `?X = p(a)`, `p(a)` is viewed as a term and `?X` is bound to it. Likewise, in `?X = a[m->v]`, the F-molecule is evaluated to its object value (which is `a`) and then unified with `?X`. To bind variables to formulas instead, $\mathcal{F}$LORA-*2* provides a `meta-unification` operator, $\sim$. This operator treats its arguments as formulas and unifies them as such. For instance, `?X` $\sim$ `a[m->v,k->?V]` binds `?X` to the F-molecule `a[m->v,k->?V]` and `a[m->v,k->?V]` $\sim$ `?X[?M->v,k->p]` unifies the two molecules by binding `?X` to `a`, `?M` to `m`, and `?V` to `p`.

Meta-unification is very useful when it is necessary to find out the module in which a particular formula lives. For instance,

```
flora2 ?- ?X@?M ~ a[b->c]@foo.
```

would bind `?X` to the formula `a[b->c]`, `?M` to the module of `?X`. Note that in meta-unification the variable `?X` in the idiom `?X@?M` or `?X@foo` is viewed as a meta-variable that is bound to a formula. More subtle examples are

```
flora2 ?- ?X ~ f(a), ?X ~ ?Y@?M.
flora2 ?- f(a)@foo ~ ?Y@?M.
```

`?M` is bound to the current module in the first query and `foo` in the second one. `?Y` is bound to the (internal representation of the) HiLog formula `f(a)@ _@` in the first query and `f(a)@foo` in the second — *not* to the HiLog term `f(a)`!

Another subtlety has to do with the scope of the module specification. In $\mathcal{F}$LORA-*2*, module specifications have scope and inner specifications override the outer ones. For instance, in

```
..., (abc@foo, cde)@bar, ...
```

the term `abc` is in module `foo`, while `cde` in module `bar`. This is because the inner module specification, `@foo`, overrides the outer specification `@bar` for the literal in which it occurs (*i.e.*, `abc`). These scoping rules have subtle impact on literals that are computed dynamically at run time. For instance, consider

```
flora2 ?- ?X@?M ~ a[b->c]@foo, ?X@bar.
```

Because `?X` gets bound to `a[b->c]@foo`, the literal `?X@bar` becomes the same as `(a[b->c]@foo)@bar`, *i.e.*, `a[b->c]@foo`. Thus, both of the following queries succeed:

```
flora2 ?- ?X@?M ~ a[b->c]@foo, ?X@bar ~ a[b->c]@foo.
flora2 ?- ?X@?M ~ a[b->c]@foo, ?X@?N ~ a[b->c]@foo.
```

Moreover, in the second query, the variable `?N` is *not* bound to anything because, as noted before, the literal `?X@?N` becomes `(a[b->c]@foo)@?N)` at run time and, due to the scoping rules, is the same as `a[b->c]@foo`.

12.2 Reification

It is sometimes useful to be able to treat $\mathcal{F}$LORA-*2* molecules and predicates as objects. For instance, consider the following statement:

```
Tom[believes-> Alice[thinks->floraProgramming:coolThing]].
```

The intended meaning here is that one of Tom's beliefs is that Alice thinks that programming in $\mathcal{F}$LORA-*2* is a cool thing. Unfortunately, this is incorrect, because, as stated, the above formula has a different meaning:

```
Tom[believes-> Alice].
Alice[thinks->floraProgramming:coolThing].
```

That is, Tom believes in Alice and Alice thinks that $\mathcal{F}$LORA-*2* programming is cool. This is different from what we originally intended. For instance, we did not want to say that Alice likes $\mathcal{F}$LORA-*2* (she probably does, but she did not tell us). All we said was what Tom has certain beliefs about what Alice thinks. In other words, to achieve the desired effect we must turn the formula `Alice[thinks->floraProgramming:coolThing]` into an object, *i.e.*, *reify* it.

Reification is done using the operator "${...}". For instance, to say that Tom believes that Alice thinks that programming in $\mathcal{F}$LORA-*2* is a cool thing one should write:

```
Tom[believes-> ${Alice[thinks->floraProgramming:coolThing]}].
```

When reification appears in facts or rule heads, then the module specification and the predicate part of the reified formula must be bound. For instance, the following statements are illegal:

```
p(${?X@foo}) :- q(?X).
p(${q(a)@?M}).
?- insert{p(${?X@?M})}.
```

The semantics of reification in $\mathcal{F}$LORA-*2* is described in [12].

Reification of complex formulas. In $\mathcal{F}$LORA-*2*, one can reify not only simple facts, but also anything that can occur in a rule body. Even a set of rules can be reified! The corresponding objects can then be manipulated in ways that are semantically permissible for them. For instance, reified conjunctions of facts can be inserted into the database using the `insert{...}` primitive. Reified conjunctions of rules can be inserted into the rulebase using the `insertrule{...}` primitive. Reified rule bodies, which can include disjunctions, negation, and even things like aggregate functions and update operators(!), can be called as queries.

```
request[
  input       -> ${?Ticket[from->?From, to->?To, not international],
  inputAxioms -> ${(?Ticket[international] :-
                        ?Ticket[from->?From:?Country1, to->?To:?Country2],
                        ?Country1 \= ?Country2)
                  }
].

?- ?Request[input->?Input, inputAxioms->?Rules],
   insertrule{?Rules},
   ?Input.
```

In the above example, the object `request` has two attributes, which return reified formulas. The `input` attribute returns a Boolean combination of molecules, while `inputAxioms` returns a

reified rule. In general, conjunctions of rules are allowed inside the reification operator (*e.g.*, `${(rule1), (rule2)}`), where each rule is enclosed in a pair of parentheses. Such a conjunction can then be inserted (or deleted) into the rulebase using the `insertrule{...}` primitive.[8]

Reification and meta-unification. Reification should not be confused with meta-unification, although they are close concepts. A reified formula reflects the exact structure that is used to encode it, so structurally similar, but syntactically different formulas might meta-unify, but their internal representations could be very different. For instance,

```
flora2 ?- a[b->?X]@?M ~ ?Y[b->d]@foo.
```

will return *true*, because the two molecules are structurally similar and thus meta-unify. On the other hand,

```
flora2 ?- ${a[b->?X]@?M} = ${?Y[b->d]@foo}.
```

will be false, because `a[b->?Y]@?X` and `?Z[b->d]@foo` have different internal representations (even though their conceptual structures are similar), so they do not unify (using "=", *i.e.*, in the usual first-order sense). Note, however, that the queries

```
?- ${a[b->?Y]@foo} = ${?Z[b->d]@foo}.
?- ?M=foo, ${a[b->?Y]@?M} = ${?Z[b->d]@?M}.
?- a[b->?Y]@foo ~ ?Z[b->d]@foo.
?- ?M=foo, a[b->?Y]@?M ~ ?Z[b->d]@?M.
```

will all return *true*, because `a[b->?Y]@foo` and `?Z[b->d]@foo` are structurally similar — both conceptually and as far as their internal encoding is concerned (and likewise are `a[b->?Y]@foo` and `?Z[b->d]@foo`).

12.3 Meta-decomposition

$\mathcal{F}$LORA-*2* supports an extended version of the Prolog meta-decomposition operator "`=..`". On Prolog terms, it behaves the same way as one would expect in Prolog. For instance,

```
flora2 ?- ?X=p(a,?Z)@_prolog, ?X=..?Y.

?X = p(a,?_h4094)@_prolog
?Z = ?_h4094
?Y = [p, a, ?_h4094]
```

[8] In fact, Boolean combinations of rules are also allowed inside the reification operator. However, such combinations cannot be inserted into the rulebase. $\mathcal{F}$LORA-*2* does not impose limitations here, since is impossible to rule out that a knowledge base designer might use such a feature in creative ways.

The main use of the `=..` operator in $\mathcal{F}$LORA-*2* is, however, for decomposing HiLog terms or of reifications of HiLog predicates and F-logic (atomic) molecules. The meta-decomposition operator uses special conventions for these new cases.

For HiLog terms, the head of the list on the fight-hand side of `=..` has the form `hilog(HiLogPredicateName)`. For instance,

```
flora2 ?- p(a,b) =.. ?L.

?L = [hilog(p), a, b]
```

For HiLog predicates the head of the list has the form `hilog(HiLogPredicateName,Module)`. For instance,

```
flora2 ?- ${p(a,b)}@foo =.. ?L.

?L = [hilog(p,foo), a, b]
```

For non-tabled HiLog predicates, which represent actions with side-effects, the head of the list is similar except that `'%hilog'` (quoted!) is used instead of `hilog`. For instance,

```
flora2 ?- ${%p(a,b)}@foo =.. ?L.

?L = [%hilog(p,foo), a, b]
```

For F-logic molecules, the head of the list has the form `flogic(MoleculeSymbol,Module)`. The `MoleculeSymbol` argument represents the type of the molecule and can be one of the following: `->`, `*->`, `=>`, `*=>`, `+>>`, `*+>>`, `->->`, `*->->`, `:`, `::`, `boolean` (tabled Boolean methods), `*boolean` (inheritable tabled Boolean methods), `%boolean` (procedural, nontabled Boolean methods), `:=:`, `[]` (empty molecules, such as `a[]`). Here is a number of examples that illustrate the use of `=..` for decomposition of F-logic molecules:

```
$a[b->c]@foo =.. [flogic(->,foo), a, b, c]
$a[b*->c]@foo =.. [flogic(*->,foo), a, b, c]
$a[b=>c]@foo =.. [flogic(=>,foo), a, b, c]
$a[b*=>c]@foo =.. [flogic(*=>,foo), a, b, c]
$a[b+>>c]@foo =.. [flogic(+>>,foo), a, b, c]
$a[b*+>>c]@foo =.. [flogic(*+>>,foo), a, b, c]
$a[b->->c]@foo =.. [flogic(->->,foo), a, b, c]
$a[b*->->c]@foo =.. [flogic(*->->,foo), a, b, c]
$a:b@foo =.. [flogic(:,foo), a, b]
$a::b@foo =.. [flogic(::,foo), a, b]
$a:=:b@foo =.. [flogic(:=:,foo), a, b]
$a[]@foo =.. [flogic([],foo), a]
$a[p]@foo =.. [flogic(boolean,foo), p]
$a[*p]@foo =.. [flogic('*boolean',foo), a, p]
$a[%p]@foo =.. [flogic('%boolean',foo), a, p]
```

The `=..` operator is bi-directional, which means that either one or both of its arguments can be bound. For instance,

```
flora2 ?- ?X =.. [flogic('*boolean',foo),a ,p].

?X = ${a[*p]@foo}
```

At present, the =.. operator does not do reified terms that represent aggregate operators, update predicates, and other $\mathcal{F}$LORA-*2* statements that are not part of the extended F-logic or HiLog syntax. In such cases, the query simply fails. In the future, `=..` might be extended to some of these additional classes of terms.

The original Prolog's `=..` is also available using the idiom `(... =.. ...)@_prolog`. This is rarely used, however. One might use this when the term to be decomposed is known to be a Prolog terms (in this case the Prolog's operator will run slightly faster) or if one wants to process the Prolog terms into which $\mathcal{F}$LORA-*2* literals are encoded internally (which is probably hardly ever necessary).

12.4 Passing Parameters between $\mathcal{F}$LORA-*2* and Prolog

The native HiLog support in $\mathcal{F}$LORA-*2* causes some tension when crossing the border from one system to another. The reason is that $\mathcal{F}$LORA-*2* terms and Prolog terms have different internal representation. Even though XSB supports HiLog (according to the manual, anyway), this support is incomplete and is not integrated well into the system — most notably into the XSB module system. As a result, XSB does not recognize terms passed to it from $\mathcal{F}$LORA-*2* as HiLog terms and, thus, many useful primitives will not work correctly. (Try `?- writeln(foo(abc))@_prolog` and see what happens.)

To cope with the problem, $\mathcal{F}$LORA-*2* provides a primitive, `p2h{?Plg,?Hlg}`, which does the translation. If the first argument, `?Plg`, is bound, the primitive binds the second argument to the Hilog representation of the term. If `?Plg` is already bound to a Hilog term, then `?Hlg` is bound to the same term without conversion. Similarly, if `?Hlg` is bound to a HiLog term, then `?Plg` gets bound to the Prolog representation of that term. If `?Hlg` is bound to a non-HiLog term, then `?Plg` gets bound to the same term without conversion. In all these cases, the call to `p2h{...}` succeeds. If `both` arguments are bound, then the call succeeds if and only if

- `?Plg` is a Prolog term and `?Hlg` is its HiLog representation.
- Both `?Plg` and `?Hlg` are identical Prolog terms.

Note that if both `?Plg` and `?Hlg` are bound to the same *HiLog term* then the predicate *fails*. Thus, if you type the following queries into the $\mathcal{F}$LORA-*2* shell, they both succeed:

```
flora2 ?- p2h{?X,f(a)}, p2h{?X,?X}.
```

but the following will fail:

```
flora2 ?- p2h{f(a),?X}, p2h{?X,?X}.
flora2 ?- p2h{f(a),f(a)}.
```

The first query succeeds because `?X` is bound to a prolog term, and by the above rules `p2h{?X,?X}` is supposed to succeed. The second query fails because `?X` is bound to a HiLog term and, again by the above rules, `p2h{?X,?X}` is supposed to fail. The reason why the last query fails is less obvious. In that query, both occurrences of `f(a)` are HiLog terms, as are all terms that appear in a *FLORA-2* program (unless they are marked with `@_prolog` or `@_prologall` module designations). Therefore, again by the rules above the query should fail.

One should not try to convert certain Prolog terms to HiLog and expect them to be the same as similarly looking *FLORA-2* terms. In particular, this applies to reified statements. For instance, if `?X = $a[b->c]` then `?- p2h{?X,?Y}, ?Y = $a[b->c]` is not expected to succeed. This is because `p2h{...}` does not attempt to mimic the *FLORA-2* compiler in cases where conversion to HiLog (such as in case of reified statements) makes no sense. Doing so would have substantially increased the run-time overhead.

Not all arguments passed back and forth to Prolog need conversion. For instance, `sort/2`, `ground/1`, `compound/1`, and many others do not need conversion because they work the same for Prolog and HiLog representations. On the other hand, most I/O predicates require conversion. *FLORA-2* provides the `io` library, described in Section 29, which provides the needed conversions for the I/O predicates.

Another mechanism for calling Prolog modules, described in Section 11.7, is to use of the `@_prologall` and `@_prologall(module)` specifiers (`@_plgall` also works). These specifiers cause the compiler to include code for automatic conversion of arguments to and from Prolog representation. However, as mentioned above, such conversion is sometimes not necessary and the use of `@_prologall` might incur unnecessary overhead.

13 Negation

FLORA-2 supports two kinds of negation: the usual Prolog's *negation as failure* [5] and negation based on *well-founded semantics* [9, 10]. Both types of negation are compiled into clauses that invoke the corresponding operators in Prolog.

We should remark that originally, the term "negation as failure" was used to denote the treatment of negation in Prolog. This style of negation is unsatisfactory in many respects because it does not have a model-theoretic characterization and because operationally it often leads to infinite loops. To overcome this problem, several different semantics were introduced, including the aforesaid well-founded semantics. At some point later, the meaning of the term negation as failure was broadened to refer to the class of all these forms of negation. When ambiguity may arise, we will be referring to *Prolog-style negation as failure*.

13.1 Two Operators for Negation

Prolog-style negation as failure is specified using the operator `\+`. Negation based on the well-founded semantics is specified using the operator `not`. The well-founded negation, `not`, applies to predicates that are tabled (i.e., predicates that do not have the `%` prefix which will be discussed in detail in Section 17) or to F-molecules that do not contain procedural methods (i.e., methods that are prefixed with a `%`).

The semantics for negation as failure is simple. To find out whether `\+?G` is true, the system first asks the query `?- ?G`. If the query succeeds then `\+?G` is said to be satisfied. Unfortunately, this semantics is problematic. It cannot be characterized model-theoretically and in certain simple cases the procedure for testing whether `\+?G` holds may send the system into an infinite loop. For instance, in the presence of the rule `%p :- \+ %p`, the query `?- %p` will not terminate. Negation as failure is the recommended kind of negation for non-tabled predicates (but caution needs to be exercised).

The well-founded negation, `not`, has a model-theoretic semantics and is much more satisfactory from the logical point of view. Formally, this semantics uses three-valued models where formulas can be true, false, or undefined. For instance, if our program has the rule `p :- not p` then the truth value of `p` is *undefined*. Although the details of this semantics are somewhat involved [10], it is usually not necessary to know them, because this type of negation yields the results that the user normally expects. The implementation of the well-founded negation in XSB requires that it is applied to goals that consist entirely of tabled predicates or molecules. Although $\mathcal{F}$LORA-*2* allows `not` to be applied to non-tabled goals, this may lead to unexpected results. For instance, Section 18 discusses what might happen if the negated formula is defined in terms of an update primitive.

For more information on the implementation of the negation operators in XSB we refer the reader to the XSB manual.

Both `\+` and `not` can be used as operators inside and outside $\mathcal{F}$LORA-*2* molecules. For instance,

```
flora2 ?- not %p(a).
flora2 ?- \+ %p(a).
flora2 ?- not X[foo->bar, bar->foo].
flora2 ?- X[not foo->bar, bar->foo, \+ %p(?Y)].
```

are all legal queries. Note that `\+` applies only to non-tabled constructs, such as non-tabled $\mathcal{F}$LORA-*2* predicates and procedural methods.

We should warn against one pitfall however. Sometimes it is necessary to apply negation to several separate literals and write something like

```
flora2 ?- \+ (%p(a),%q(?X)).
flora2 ?- not (p(a),q(?X)).
flora2 ?- not (?X[foo->bar], ?X[bar->foo]).
```

This is incorrect however, since in this context $\mathcal{F}$LORA-*2* (and Prolog as well) will interpret `not` and `\+` as predicates with two arguments, which are likely to be undefined. The correct syntax is:

```
flora2 ?- \+ ((%p(a),%q(?X))).
flora2 ?- not ((%p(a),%q(?X))).
flora2 ?- not ((?X[foo->bar], ?X[bar->foo])).
```

i.e., an additional pair of parentheses is needed to indicate that the sequence of literals form a single argument.

13.2 True vs. Undefined Formulas

The fact that the well-founded semantics for negation is three-valued brings up the question of what exactly does the success or failure of a call means. Is undefinedness covered by a success or by failure? The way this is implemented in XSB is such that a call to a literal, P, succeeds if and only if P is true *or* undefined. Therefore, it is sometimes necessary to be able to separate true from undefined facts. In $\mathcal{F}$LORA-*2*, this separation is accomplished with the $\mathcal{F}$LORA-*2* primitives `true{`*Goal*`}` and `undefined{`*Goal*`}`. For good measure, the primitive `false{`*Goal*`}` is also thrown in. For instance,

```
a[b->c].
e[f->g] :- not e[g->g].
?- true{a[b->c]}.

Yes

?- unknown{e[f->g]}.

Yes

?- false{k[l->m]}.

Yes
```

It should be noted that the primitives `true{...}` and `unknown{...}` can be used only in the top-level queries. Otherwise, correctness of the result is not guaranteed. The expression `false{Goal}` is equivalent to `not Goal`, and can be used anywhere.

13.3 Unbound Variables in Negated Goals

When negation (either `\+` or `not`) is applied to a non-ground goal, one should be aware of the following peculiarity. Consider `\+ ?Goal`, where `?Goal` has variables that are not bound. As mentioned before, `\+ ?Goal` is evaluated by posing `?Goal` as a query. If for *some* values of for the variables in `?Goal` the query succeeds, then `\+ ?Goal` is false; it is true only if for *all* possible substitutions for the variables in `?Goal` the query is false (fails). Therefore `\+ ?Goal` intuitively means $\forall$?`Vars` $\neg$?`Goal`, where `?Vars` represents all the nonbound variables in `?Goal`. The well-founded negation has the same flavor: if `?Goal` is non-ground then `not ?Goal` means $\forall$?`Vars` $\neg$?`Goal`.

Of course, one should keep in mind that since neither \+ nor not is a classical negation, none of the above formulas is actually equivalent to ∀?Vars ¬ ?Goal, if ¬ is understood as classical negation. A more precise meaning is that not ?Goal is true if and only if for every ground instance ?Goal′ of ?Goal, the literal not ?Goal′ is true in the well-founded semantics. Similarly, \+ ?Goal evaluates to true if and only if for every ground instance ?Goal′ of ?Goal, the query \+ ?Goal′ succeeds according to the negation-as-failure semantics.

To illustrate this, consider the following example:

```
p(a,b).
q(?X,?Y) :- not p(?X,?Y).
flora ?- q(?X,?Y).
```

When not p(?X,?Y) is called in the query evaluation process, the variables are unbound, so for the query to return a positive answer, the literal p(t,s) should be false for every possible terms t and s. Since p(a,b) is true, our query q(?X,?Y) fails. In contrast, the query

```
flora2 ?- q(b,?Y).
```

will succeed because this will cause the query not p(b,?Y) to be evaluated. But this query will return positive answer because p(b,?Y) is false for all ?Y. Note that even when the query succeeds the unbound variable that occurs in the scope of the negation operator remains unbound:

```
flora2 ?- not p(b,?Y).

?Y = _h1747

1 solution(s) in 0.0000 seconds on speedy.foo.org
```

14 Inheritance

In general, inheritance means that attribute and method specifications for a class are propagated to the subclasses of that class and to the objects that are instances of that class.

F-logic (and *FLORA-2*) distinguishes between attributes and methods that can inherit values from superclasses and those that do not. The syntax that we have seen so far in this manual applies to *non-inheritable* attributes only. *Inheritable attributes* are declared using the *=> style arrow and defined using the *-> style arrow. Note that while -> typically occurs in facts and rules that define the properties of individual objects, *-> normally occurs in definitions of classes.

Non-inheritable attributes sometimes correspond to what is known as *class variables* in traditional object-oriented languages. For example, the attribute "average age" would be such an attribute for class person. It does not make sense to propagate this attribute and its value to the instances of that class, because the concept of an average age does not apply to individual people. Similarly, it does not make sense to propagate this attribute to subclasses, such as student, because the average age of students is likely to be different from that of persons, and therefore the inherited value would be of no use. In *FLORA-2*, we would specify such a fact as follows:

```
person[avg_age -> 40].
```

Similarly, attributes that typically refer to individuals are better specified as non-inheritable, because normally there is nothing to inherit these attributes to:

```
John[age -> 30].
```

Inheritable attributes typically define default properties of the objects in a class, such as

```
british[nativeLanguage *-> 'English'].
```

If `John:british` is true, then, without evidence to the contrary, we can derive `John[nativeLanguage -> 'English']`. If we are also told that `scottish::british`, *i.e.*, Scottish people are also British, then we can derive (again, in the absence of a counter-evidence) that `scottish[nativeLanguage *-> 'English']`.

Note that the form of the arrow, `->`, mutates when an attribute or a method is inherited to a member of a class. In general, an inheritable attribute is inherited to a subclass as an inheritable attribute and to a member of a class as a non-inheritable attribute. In other words, `*=>` and `*->` do not change when inherited to subclasses, but they change to `=>` and `->`, respectively, when inherited to class members.

14.1 Structural vs. Behavioral Inheritance

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

Behavioral inheritance is much more complicated. It is *non-monotonic* in the sense that addition of new facts might falsify previously true facts.

The following is a $\mathcal{F}$LORA-*2* program for the classical `Royal Elephant` example:

```
elephant[color*=>color].
royal_elephant::elephant.
clyde:royal_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 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, it must be white, while being an elephant it must be gray. The conventional wisdom in object-oriented languages, however, is that inheritance from more specific classes must take precedence. Thus, we must withdraw our earlier conclusion that `clyde` is gray and infer that he is white:

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

Nonmonotonicity also arises due to multiple inheritance. The following example, known as the Nixon Diamond, illustrates the problem. Let us assume the following knowledge base:

```
republican[policy *-> nonpacifist].
quaker[policy *-> pacifist].
nixon:quaker.
```

Since Nixon is a Quaker, we can derive `nixon[policy -> pacifist]` by inheritance from the second clause. Let us now assume that the following information is added:

```
nixon:republican.
```

Now we have a conflict. There are two conflicting inheritance candidates: `policy *-> pacifist` and `policy *-> nonpacifist`. In $\mathcal{F}$LORA-*2*, such conflicts cause previously established inheritance to be withdrawn and the value of the attribute `policy` becomes undefined for object `nixon`.[9]

Behavioral inheritance in F-logic is discussed at length in [13]. The above non-monotonic behavior is just the tip of an iceberg. Much more difficult problems arise when inheritance interacts with regular deduction. To illustrate, consider the following program:

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

[9] This behavior can be altered by adding additional rules. For instance, one could define a predicate `hasPriority` and then define

```
?Obj[policy->?P] :- ?Obj:?Class, ?Class[policy*->?P], not hasPriority(?AnotherClass,?Class).
```

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, we must necessarily have `a[m->{c,d}]`. So, the value of attribute `m` is not really the one 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 similar situation (with similarly conflicting conclusions) arises when the class hierarchy is not static. For instance,

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

If we inherit `a[m ->c]` from `b` (which seems to be OK in the beginning, because nothing overrides this inheritance), then we derive `a:d`, *i.e.*, we get the following: `a:d::b`. This means that *now* `d` seems to be negating the reason why `a[m ->c]` was inherited in the first place. Again, we can either undo the inheritance or adopt the principle that inheritance is never undone.

A semantics that favors the second interpretation was proposed in [8]. 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 the underlying Prolog engine. It is also unsatisfactory in many respects because it is not based on a model-theory. *F*LORA-*2* uses a different, more cautious semantics for inheritance, which favors the first interpretation above.

Details of this semantics are formally described in [13]. Under this semantics, `clyde` will still inherit color white, but in the other two examples `a[m->c]` is *not* inherited. The basic intuition can be summarized as follows:

1. Method definitions in subclasses override the definitions that appear in the superclasses.
2. In case of a multiple inheritance conflict, the result of inheritance is undefined. More precisely, *F*LORA-*2* is based on a three-valued logic and in this case the truth value is "unknown."
3. Inheritance from the same source through different paths is *not* considered a multiple inheritance conflict. For instance, in

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

Even though we derive `c[m*->f]` and `d[m*->f]` by inheritance, these two facts can be further inherited to the object `a`, since they came from a single source `e`.

On the other hand, in a similar program

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

inheritance does not take place (the truth value of `a[m->f]` is "undefined"), because the two inheritance candidates, `c[m*->f]` and `d[m*->f]`, are considered to be in conflict.

Note that in the last example one might argue that even if we did inherit both facts to `a` there would be no discrepancy, because in both cases the values of the attribute `m` agree with each other. However, $\mathcal{F}$LORA-*2* views this agreement accidental, as it depends on the data currently stored in the database. Had one of the values changed to, say, `d[m->g]`, there would be a conflict.

4. At the level of methods of arity > 1, a conflict is considered to have taken place if there are two non-overwritten definitions of the same method attached to two different superclasses. When deciding whether a conflict has taken place we disregard the arguments of the method. For instance, in

   ```
   a:c. c[m(k)*->f].
   a:d. d[m(u)*->f].
   ```

   a multiple inheritance conflict has taken place even though in one case the method `m` is applied to object `k`, while in the other it is applied to object `u`.

   On the other hand,

   ```
   a:c. c[m(k)*->f].
   a:d. d[m(k,k)*->f].
   ```

   do not conflict, because `m/1` in the first case is a different method than `m/2` in the second. Similarly,

   ```
   a:c. c[m(k)()*->f].
   a:d. d[m(u)()*->f].
   ```

   are not considered in conflict because here it is assumed that the method names are `m(k)` and `m(u)`, which are distinct names.

In the examples that we have seen so far, path expressions used only non-inheritable attributes. Clearly, there is no reason to disallow inheritable attributes in such expressions. To distinguish inheritable attributes from non-inheritable ones, $\mathcal{F}$LORA-*2* uses the symbol `!` in its path expressions. For instance,

```
clyde!color          means: some ?X, such that clyde[color*->?X]}.
```

14.2 Code Inheritance

The type of behavioral inheritance defined in the previous subsection is called *value inheritance*. It originates in Artificial Intelligence, but is also found in modern main stream object-oriented languages. For instance, it is related to inheritance of static methods in Java. With this inheritance, one would define a method for a class, e.g.,

```
cl[attr->14].
cl[foo(?Y) *-> ?Z] :- cl[attr->?V], ?Z is ?V+?Y.
```

Every member of this class will then inherit exactly the same definition of `foo`, which refers to the class property `attr`. Since the method definition has no way to refer to the instances on which it is invoked, this method yields the same result for all class instances. One way to look at this is that class instances do not really inherit the definition of the method. Instead, the method is invoked in the context of the class where it is defined and then the computed value is inherited down to all instances (provided that they do not override the inheritance). So, if `a:cl` and `b:cl` then `a.foo(4)` and `b.foo(4)` will return exactly the same value, 18.

A more common kind of methods is called *instance methods* in Java. In this case, the method definition refers to instances of the class in whose context the method is supposed to be invoked. The invocation takes place as follows. First, a class member inherits the *code* of the method. Then the code is executed in the context of that class member.

In F-logic this kind of inheritance is called *code inheritance* and was studied in [14]. Code inheritance is not yet supported by $\mathcal{F}$LORA-*2*. However, with some loss of elegance and extra work, code inheritance can often be simulated using value inheritance. The method consists of three steps.

1. Define desired methods for all appropriate objects irrespective of classes. Definitions of these methods are the ones to be inherited using simulated code inheritance.
2. Define attributes whose values are the names of the methods defined in (1). These attributes will be subject to value inheritance.
3. Specify how the "real" methods in (1) represented by the "fake" methods in (2) are to be invoked on class instances.

We illustrate this process with the following example. First, assume the following database:

```
aa:c1.
bb:c2.
c1::c2.
aa[attr1->7, attr2->2].
bb[attr1->5, attr2->4].
```

We are going to show how code is inherited from `c2` to `bb`. In an attempt to inherit the same code from `c2` to `aa`, it will be overwritten by code from `c1` and the latter will be inherited by `aa`.

```
// method foo/1 defined for every instance
?X[foo(?Y) -> ?Z] :- ?X[attr1->?V], ?Z is ?V+?Y.
// method bar/1 defined for every instance
?X[bar(?Y) -> ?Z] :- ?X[attr2->?V], ?Z is ?V*?Y.
```

Unlike Java, the above code is not really local to any class, and this is one aspect in which simulation of code inheritance by value inheritance is inelegant. Next we define `meth` — the method whose value inheritance will simulate the inheritance of code of `foo` and `bar`.

```
c1[dispatch(meth) *-> bar].
c2[dispatch(meth) *-> foo].
```

Clearly, the object `bb` will inherit `dispatch(meth)->foo` from `c2`, while the object `aa` will inherit `dispatch(meth)->bar` from `c1`; inheritance from `c2` is overwritten.

Next, we define how methods are to be invoked in a way that resembles code inheritance:

```
?X[?M(?Y) -> ?Z] :- ?X[dispatch(?M)->?RealMeth], ?X[?RealMeth(?Y) -> ?Z].
```

When `?M` is bound to a particular method, say `meth`, and this method is invoked in the context of a class instance, `?X`, the invocation `?X[meth(?Y)->?Z]` first computes the value of the attribute `dispatch(meth)`, which gives the name of the actual method to be invoked. The value of the `dispatch(meth)` attribute (represented by the variable `?RealMeth`) is obtained by value inheritance. As explained above, this value is `foo` when `?X` is bound to `bb` and `bar` when `?X = aa`. Finally, the real method whose name is obtained by value inheritance is invoked in the context of the class instance `?X`. One can easily verify the following results:

```
flora2 ?- aa[meth(4) -> ?Z].

?Z = 8

flora2 ?- bb[meth(4) -> ?Z].

?Z = 9
```

This is exactly what would have happened in Java if `aa` inherited the instance method whose code is equivalent to the definition of `bar/1` and if `bb` inherited the code of `foo/1`.

15 Custom Module Semantics

*F*LORA-*2* enables the user to choose the appropriate semantics for any user module. This is done with the help of the following directive:

```
:- setsemantics{Option1, Option2, ...}
```

Three kinds of options are allowed:

Equality: `equality(none)`, `equality(basic)`; `equality(none)` is the default.

Inheritance: `inheritance(none)`, `inheritance(flogic)`; `inheritance(flogic)` is the default.

Custom: `custom(none)`, `custom(`*`filename`*`)`; `custom(none)` is the default.

These options are described in more detail in the following subsections. Within each group only one choice can be present or else an error will result. It is not required that all options be present — defaults are substituted for the missing options.

The compiler directive described above determines the initial semantics used by the module in which the instruction occurs. However, it is also possible to change the semantics at run time using the *executable directive*:

```
?- setsemantics{Option1, Option2, ...}
```

Note the use of `?-` here: the symbol `:-` in the first directive designates the directives that are used at compile time only. Executable directives, on the other hand, can occur in any query or rule body. It is also possible for one module to change the semantics in another module. Typically this is needed when one module creates another. In this case the new module is created with the default semantics, and the `setsemantics` executable directive makes it possible to change the semantics of such a module. Here is an example:

```
?- setsemantics{equality(basic), custom('a/b/c')}.
```

The order of the options in the directive does not matter.

Changing module semantics — precautions. Changing module semantics on the fly at run-time is a rather drastic operation. It is therefore *not* recommended to do this in the body of a rule, especially if the rule defines a tabled HiLog predicate or an F-logic molecule. The only safe way to execute `setsemantics` is in a query at the top level. For instance,

```
?- setsemantics{...}.
```

15.1 Equality Maintenance

User-defined equality. $\mathcal{F}$LORA-*2* users can define equality *explicitly* in the source program using the predicate `:=:`, e.g.,

```
John:=:Batman.
?X:=:?Y :- ?X[similar->?Y].
```

Once two oids are established to be equal with respect to `:=:`, whatever is true of one object is also true of the other. Note that `:=:` is different from the built-in `=`. The latter is a predefined primitive, which cannot occur in the facts or in the rule head. Since `=` is understood as unification, ground terms can be =-equal only if they are identical. Thus, `a=a` is always true and `a=b` is always false. In contrast, the user can assert a fact such as `a:=:b`, and from then on the object `a` and the object `b` are considered the same (modulo the equality maintenance level, which is described below).

Equality maintenance levels. Once an equality between terms is derived, this information may need to be propagated to all F-logic structures, including the subclass hierarchy, the ISA hierarchy, etc. For instance, if `x` and `y` are equal, then so must be `f(x)` and `f(y)`. If `x:a` has been previously derived then we should now be able to derive `y:a`, etc. Although equality is a powerful feature, its maintenance can slow the program down quite significantly. In order to be able to eat the cake and have it at the same time, *FLORA-2* allows the user to control how equality is handled. by providing the following three compiler directives:

```
:- setsemantics{equality(none)}.  (default)
:- setsemantics{equality(basic)}.
```

The first directive, `setsemantics{equality(none)}`, does not maintain any equality and `:=:` behaves similarly to the regular unification operator but additional facts and rules can be inserted to augment the definition of this predicate. Under this semantics, `:=:` is not transitive and the special congruence properties of equality are not supported (for instance, `p(a)` and `a:=:b` do not imply `p(b)`). The directive `setsemantics{equality(basic)}` guarantees that `:=:` obeys the usual rules for equality, i.e., transitivity, reflexivity, symmetry, and (limited) substitution.

If a *FLORA-2* module does not define facts of the form `a:=:b`, which involve the equality predicate `:=:`, then the default equality maintenance level is `none`. If the program does define such facts, then the default equality maintenance level is `basic`, because it is assumed that the use of `:=:` in the program is not accidental. In any case, the explicit `equality(...)` option overrides the default.

Note that even if the module might have path expression in the head, the default equality level is still `none` (unless `:=:` is used). The reason for this is that such path expressions do not always require equality maintenance, so the user has to request it explicitly. For instance, if in the above example we never insert `John[mother->Sally]` then no equality maintenance will be required even if the program defines the fact `John.mother[father->Bob]`, as above. However, if this fact is inserted, then the equality maintenance level appropriate for this case is `flogic` (`basic` will not be sufficient).

Locality of equality. Equality in *FLORA-2* is always local to the module in which it is derived. For example, if `a:=:b` is derived by the rules in module `foo` then the query

```
flora2 ?- (a:=:b)@foo.
```

will succeed, but the query

```
flora2 ?- (a:=:b)@bar.
```

will fail (unless, of course, `a:=:b` is also derived by the rules in module `bar`).

Since equality information is local to each module, the directives for setting the equality level affect only the particular user modules in which they are included. Thus, equality can be treated differently in different modules, which allows the programmer to compartmentalize the performance problem associated with equality and, if used judiciously, can lead to significant gains in performance.

Run-time changes to the equality maintenance level. In *FLORA-2*, the desired level of equality maintenance can also be changed at run time by executing a goal such as

```
?- setsemantics{equality(basic)}.
```

Furthermore, *FLORA-2* allows one user module to set, at run time, the level of equality maintenance in another user module:

```
?- setsemantics{equality(basic)}@foobar.
```

This might be useful for *dynamic* modules, *i.e.*, modules that are not associated with any files and whose content is generated completely dynamically. (See Section 18.)

Using the preprocessor to avoid the need for equality maintenance. One final advice regarding equality. In many cases, programmers tend to use equality as an aliasing technique for long messages, numbers, etc. In this case, we recommend to use the preprocessor commands, which achieve the same result without loss of performance. For instance,

```
#define YAHOO  'http://yahoo.com'

?- YAHOO[fetch -> ?X].
```

Assuming that `fetch` is a method that applies to strings that represent WWW sites and that fetches the corresponding Web pages, the above program will fetch the page at the Yahoo site, because *FLORA-2* compiler will replace YAHOO with the corresponding string that represents a URL.

Limitations of equality maintenance in *FLORA-2*. The implementation of equality in *FLORA-2* supports only a limited version of the *congruence axiom* due to the overhead associated with such an implementation. A congruence axiom states that if α `=` β then β can be substituted for any occurrence of α in any term. For instance, $f(x, \alpha) = f(x, \beta)$. In *FLORA-2*, however, the query

```
a :=: b.
?- g(a) :=: g(b).
```

will fail. However, equal terms can be substituted for the arguments of F-logic molecules and HiLog predicates. For instance, the queries

```
a:=:b.
a[f->c].
p(a,c).
?- b[f->c].
?- p(b,c).
```

will succeed.

15.2 Choosing an Inheritance Semantics

As mentioned earlier, the `setsemantics` directive accepts two options: `inheritance(none)` and `inheritance(flogic)`. The default is `flogic`; this type of inheritance is described in Section 14.

With `inheritance(none)`, behavioral inheritance is turned off in the corresponding module. This can significantly improve performance in cases when inheritance is not needed.

Note that `inheritance(none)` does *not* turn off inheritance of signatures. Inheritance of signatures can be used for run-time type checking and it makes no good sense to disable it. Preserving inheritance of signatures does not affect the performance either.

15.3 Ad Hoc Custom Semantics

The `setsemantics` directive allows the user to include additional axioms that define the semantics of a particular module. These axioms should be stored in a file and included into the module using the compiler or executable directive

```
:- setsemantics{custom(filename)}.
```

However, the default is `custom(none)`[10] To take advantage of this feature, the user must write the axioms using the same API that is used for $\mathcal{F}$LORA-*2* trailers, which are located in the `closure` directory of the distribution. This API will be described at a later date.

15.4 Querying Module Semantics

In addition to the ability to change the semantics of a module, $\mathcal{F}$LORA-*2* also lets the user *query* the semantics used by any given module through the `semantics` primitive. The syntax is similar to the `setsemantics`directive:

```
?- semantics{ Option1,  Option2, ...}.

?- semantics{ Option1,  Option2, ...}@modulename.
```

[10] Which implies that if the file has the name `none` then a full path name should be specified — just "none" implies no custom file.

The options are the same as in the case of the `setsemantics` directive, but variables are allowed in place of the specific semantic choices, *e.g.*, `equality(X)`. The options unify with the current semantic settings in the module, so queries such as

```
?- semantics{equality(X), custom(none)}.

?- semantics{inheritance(flogic), equality(?X), custom(?Y)}@foo.
```

are allowed. The order of the options in a `semantics`-query does not matter.

The `@`*`module`* part in the `semantics` primitive must be bound to a module name at the time the query is executed. However, it is still possible to find out which modules have any given combination of semantic options by examining every loaded module via the `_isloaded/1` builtin and then posing the desired `semantics{...}` query.

16 Cardinality Constraints

The earlier versions of F-logic made a distinction between functional and set-valued attributes and methods. The former were allowed to have only one value for any particular object and the latter could have any. In $\mathcal{F}$LORA-*2*, this dichotomy was replaced with the much more general mechanism of cardinality constraints. These constraints can be specified in signature expressions, which we have earlier used only to define types of attributes and methods. The extended syntax is as follows:

```
Cl[Meth{LowerBound:UpperBound}=>Cl2]
Cl[Meth{LowerBound:UpperBound}*=>Cl2]
```

The first signature applies to object `Cl` and to its noninheritable method `Meth`. The second expression applies to the inheritable method `Meth` of `Cl`, of the subclasses of `Cl`, and to *non*inheritable method `Meth` of the objects that belong to class `Cl`. (Recall that inheritable methods are inherited as inheritable methods to subclasses, but they become non-inheritable once they are inherited to class members.)

The lower and upper bounds in cardinality constraints can be non-negative integers, variables, or the symbol `*`. Variables can occur in signatures in rule bodies when one wants to query the bounds of the cardinality constraints and `*` means infinity.

For example,

```
c1[m{2:?X}=>c2] :- ?X=3.
```

means that the method `m` of class `c1` must have at least 2 at most 3 values. Similarly,

```
c1[m{2:*}=>c2].
```

means that `m` has at least 2 values; there is no upper bound.

We can query the specified cardinality constraints by putting variables in the appropriate places. For instance, consider the following knowledge base loaded into module `foo`:

```
C[m{3:*}*=>B].
C[m{?x:1}=>B] :- ?x=0.

v:C.
C2::C.
v2:C2.

C[m->{1,2}].
v[m->2].
C2[m*->{1,2,3}].
```

The query

```
?- ?C[?M{?L:?H}=>?]@foo.
```

will yield three solutions:

```
?C = C
?M = m
?L = 0
?H = 1

?C = v
?M = m
?L = 3
?H = *

?C = v2
?M = m
?L = 3
?H = *
```

Note that the objects `v` and `v2` are in the answer to the query because they inherited the cardinality constraint for non-inheritable version of `m` from the first clause, `C[m3:**=>B]`.

On the other hand, the query

```
?- ?C[?M{?L:?H}*=>?]@foo.
```

has two solutions:

```
?C = C
?M = m
?L = 3
?H = *
```

```
?C = C2
?M = m
?L = 3
?H = *
```

Class `C` is in the result because the constraint is specified explicitly and `C2` is in the result because it inherited the constraint from `C`.

17 FLORA-2 and Tabling

17.1 Tabling in a Nutshell

Tabling is a technique that enhances top-down query evaluation with a mechanism that remembers the calls made previously in the process. 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-2 compiler automatically tables F-molecules and HiLog predicates. If the user wants to use a non-tabled predicate, she must use a predicate name that begins with the "`%`" sign.

For instance, in the following rules, `tc/2` is tabled but `%edge/2` is not tabled.

```
tc(X,Y) :- %edge(?X,?Y).
tc(X,Y) :- %edge(?X,?Y), tc(?Y,?Z).
```

A predicate with the `%` prefix is logically unrelated to the predicate without the `%` prefix. Thus, `p(a)(b)` being true does not imply anything about `%p(a)(b)`, and vice versa.

Identifiers and variables that are prefixed with the "`%`" sign can appear only as predicate formulas, predicate names, or Boolean method names. However, a variable prefixed with "`%`" can not be a stand-alone formula, unless it is associated with a module specification. The following occurrences of "`%`" are legal

```
?- insert{%p(a)}, %?(?X).  // %? is a variable ranging over non-tabled
                          // predicate names
?- a[%b(c)], a[%?Y].       // %b and %?Y are procedural Boolean methods
?- %?X@?M ~ %p(a).          // %p - a non-tabled predicate
```

but the following are not:

```
?- p(%a).                   // %a appears as a term, not formula
?- ?X = %a.                  // %a appears as a term, not formula
```

```
?- %?X = a.                    // %?X appears as a term, not formula
?- a[%b(c)->d].               // %b is not a Boolean method
?- %?X ~ %p(a).                // %?X as a stand-alone formula
```

The first formula is illegal because `%a` occurs as a term and not as a predicate (it can be made legal by reifying the argument: `p(${%a})`). In the second and third formulas `%a` and `%?X` also appear as unreified arguments. The fourth formula is illegal because `%b(c)` is not a Boolean method. The last one is illegal because `%?X` can not be a stand-alone formula (it can be made legal by associating a module with it).

Occurrences of variables that are prefixed with `%` are treated specially. First, it should be kept in mind that `%?X` and `?X` represent the same variable. If `?X` is *already bound* to something then all both of them mean the same thing. However, `?X` itself can range not only over predicates but also terms, conjunctions/disjunctions of predicates, and even rules. In contrast, `%?X` with module specification can be bound only to non-tabled formulas and `?X` with module specification can be bound only to tabled formulas. Thus error messages will be issued for the following two queries:

```
?- ?X ~ p(a), %?X@?M ~ p(a).
?- ?X ~ a[%b], ?X@?M ~ a[%b].
```

The following query fails because `%?X` and `?X` represent the same variable: the first conjunct determines the binding for `?X`, and this binding does not match the expression on the right side of $\sim$ in the second conjunct.

```
?- %?X@?M ~ %p(a), ?X ~ p(a).
```

In the query, `?X` is bound to the non-tabled formula `%p(a)`, and this does not meta-unify with the tabled formula `p(a)`.

When a bound variable occurs with an explicit module specification, then the following rules apply:

- If the idiom `?X@module` is used, `?X` can be bound only to a tabled predicate, a tabled molecular formula, or a Hilog *term* (not a predicate). Otherwise, an error is issued. If `?X` is already bound to a tabled predicate or molecular formula, then the explicit module specification (`@module`) is discarded. When `?X` is bound to a HiLog term, e.g., `p(a)(?Z)`, `?X@module` represents the tabled predicate `p(a)(?Z)@module`.
- If the idiom `%?X@module` is used, `?X` can be bound to only a non-tabled predicate, a non-tabled molecular formula, or a Hilog *term*. If `?X` is already bound to a non-tabled predicate or molecular formula, the explicit module specification is discarded, as before. If `?X` is bound to a HiLog term, then `%?X@module` represents the non-tabled predicate `p(a)(?Z)@module`.

Due to these rules, the first query below succeeds, while the second fails and the third causes an error.

```
flora2 ?- ?X = p(a), %?X@?M ~ %p(a), ?X@?N ~ p(a)@foo.
flora2 ?- ?X ~ p(a), ?X@?M ~ p(a)@foo.
flora2 ?- ?X ~ p(a), %?X@?M ~ %p(a)@foo.
```

The first query succeeds because `?X` is bound to the term `p(a)`, which `%?X@?M` promotes to a non-tabled predicate with yet-to-be-determined module. The meta-unification that follows then binds `?M` to `main`. Similarly `?X@?N` promotes the term `p(a)` to a tabled predicate with a yet-to-be-determined module, and meta-unification binds `?N` to `foo`. The second query fails because `?X` is already bound to a tabled predicate and therefore `?X@?M` represents `p(a)@main`, which does not meta-unify with `p(a)@foo`. The third query gives an error because `?X` is bound to a tabled predicate, while `%?X@?M` expects a non-tabled predicate or a HiLog term.

When `?X` and `%?X` occur with explicit module specifications and are *unbound* then the occurrences of `%?X` indicate that `?X` is expected to be bound to predicate names, Boolean method names, or predicate/molecular formulas that correspond only to non-tabled methods or predicates. Likewise, an occurrence of an unbound `?X` indicates that `?X` is expected to be bound to predicate names or predicate/molecular formulas that correspond to tabled methods or predicates.

%-prefixed variables and meta-programming. In meta-unifications, update operations and the `clause` construct, variables that are prefixed with a "`%`" to indicate non-tabled occurrences must have explicit module specifications. An unprefixed variable without a module specification, such as `?X`, can meta-unify with both tabled and non-tabled predicates. However, when an explicit module specification is given, such as in `?X@main`, unprefixed variables can be bound only to tabled predicates. For example, all of the following queries succeed without errors.

```
?- ?X ~ %p(a).
?- ?X ~ p(a).
?- ?X ~ a[b->c]@foo.
?- ?X ~ a[%b]@?M.
?- ?X@?M ~ p(a).
?- %?X@foo ~ a[%b]@?M.
```

In the context of update operations, FLORA-2 uses the same rules for variables of the form `%?X` and `?X`. Therefore, the following operations will succeed:

```
?- insert{p(a),%q(b)}.      // Yes
?- delete{?X@_@}.           // Yes, with ?X is bound ${p(a)}
?- delete{%?X@_@}.          // Yes, with ?X is bound ${%q(b)}
?- insert{p(a),%q(b)}.      // Yes
?- delete{?X}.              // Yes, ?X is bound to ${p(a)} or ${%q(b)}
```

These rules also apply to queries issued against rule bases using the `clause` primitive (see Section 20 for the discussion of this primitive) or to deletion of rules with the `deleterule` primitive.

```
?- insertrule{p(?X) :- q(?X)}.
```

```
?- insertrule{%t(?X) :- %r(?X)}.
?- insertrule{pp(?X) :- q(?X), %r(?X)}.
?- clause{?X,?Y}.          // all three inserted rules above would be retrieved
?- clause{%?X@_@,?Y}.      // ?X = %t(?_var) and ?Y = %r(?_var)
?- clause{?X@_@,?Y@_@}.    // ?X = p(?_var) and ?Y = q(?_var)
?- clause{?X@_@,?Y}.       // the first and the third rules would be retrieved
```

It is important to keep in mind that Prolog does not reorder F-logic molecules and predicates during joins. Instead, all joins are performed left-to-right. Thus, program clauses must be written 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 default notion of similarity used by XSB is fairly weak, and many unnecessary recomputations might result. Recently, a new technique, called *subsumptive tabling*, has been implemented in XSB. It is known that subsumptive tabling can speed up certain queries by an order of magnitude. A future version of FLORA-2 might take advantage of this technique.

17.2 Discarding Information Stored in Prolog Tables

When Prolog (and FLORA-2) evaluate a program, all tabled predicates are partially materialized and all the computed tuples are stored in Prolog tables. Thus, if you change the underlying set of facts (via insert and delete operations), the existing tables must be discarded in order to allow Prolog to recompute the results. We discuss updates and the problems caused by tabling Section 18.

There are two ways to discard tabled information in FLORA-2. One, and the safest way is to use the operator `refresh{...}`. Inside the braces you list the calls for which you want to discard table information. For instance,

```
flora2 ?- refresh{p(a,?X), ?X[meth(?_)->b]}.
```

will discard any tabling information that is related to `p(a,?X)` and `?X[meth(?_)->b]`. To affect the tables in another module, attach the module name to the corresponding literals. For instance,

```
flora2 ?- refresh{(p(a,?X), ?X[meth(?_)->b])@foo}.
```

Sometimes it may be desirable to discard *all* table information in the current run of the program. This can be done by issuing the query `?- abolish_all_tables/0` described in the XSB manual. However, this should be done with great caution, because `abolish_all_tables/0` is not a safe query (it can crush XSB!).

If you really need to use `abolish_all_tables/0`, it *cannot* be used in the following cases:

1. in the body of a rule that defines an object attribute or method
2. in the body of a rule that defines a tabled HiLog predicate

This is because internally the above entities are represented using tabled predicates. Execution of `abolish_all_tables/0` in the body of such a rule would destroy the table for the predicate being computed by that rule.

More generally, no tabled predicate or object molecule of the above sort can depend via rules — directly or indirectly — on `abolish_all_tables/0`. However, it is safe to use this predicate in the body of a rule that defines a procedural method (defined next).

Note: Neither `refresh{...}` nor `abolish_all_tables` can occur under the scope of the negation operator `not` (either directly or indirectly).

17.3 Procedural Methods

Because tabling is not integrated with the update mechanism in Prolog, it can have undesirable effect on predicates with non-logical "side effects" (e.g., writing or reading a file) and predicates that change the state of the database. If a tabled predicate has a side effect, the first time the predicate is called the side effect is performed, but the second time the call simply returns with success or failure (depending on the outcome of the first call), because Prolog will simply look it up in a table. Thus, if the predicate is intended to perform the side effect each time it is called, it will not operate correctly.

Object-oriented programs often rely on methods that produce side effects or make updates. In FLORA-2 we call such methods *procedural*. Because by default FLORA-2 tables everything that looks like an F-molecule, these procedural methods are potentially subject to the aforesaid problem.

To sidestep this problem, FLORA-2 introduces a new syntax to identify procedural methods — by allowing the "`%`" sign in front of a procedural method. For instance, the following rule defines an output method that, for every object, writes out its oid:

```
?O[%output] :- write(?O)@_prolog.
```

Like boolean methods, procedural methods can take arguments, but do not return any values. The only difference is that procedural methods are *not* tabled, while boolean methods are.

17.3.1 Procedural Signatures

Procedural methods can have signatures like other kinds of methods. For noninheritable Boolean methods, signatures are specified as follows:

```
Class[=>%Meth]
```

FLORA-2 does not support inheritable procedural methods at present, but the syntax permits signatures for such methods (which are just ignored):

```
C[*=>%Meth]
```

17.4 Operational Semantics of FLORA-2

Although FLORA-2 is a declarative language, it provides primitives, such as input/output, certain types of updates, cuts, etc., which have no logical meaning. In such cases, it is important to have an idea of the *procedural semantics* of FLORA-2. This procedural semantics is essentially the same as in XSB and when no tabled predicates or F-logic molecules are involved, the behavior is the same as in Prolog. However, when tabled HiLog predicates or F-logic molecules (other than procedural methods) are used the programmer must have some understanding of the way XSB evaluates tabled predicates.

XSB has two configuration modes that affect tabled predicates: *batched* and *local* (the default). These modes affect *scheduling*, *i.e.*, the order in which answers to the literals in a rule body are computed. (The current release does not work under batched scheduling, so reading on is even more important to understand the flow of control under local evaluation.)

Under the batched scheduling, the behavior is similar to that of Prolog. Under the local scheduling, answers to the entire clique of inter-dependent predicates is computed before the computation proceeds to the next literal in a rule body. The following little program illustrates the difference:

```
a:b.
d:b.
c:b.

?X[foo(?Y)] :- ?X:?Y, writeln(?X)@_prolog.
%q(?X,?Y) :- ?X:?Y, writeln(?X)@_prolog.

?- ?X[foo(?Y)], writeln(done)@_prolog.
?- %q(?X,?Y), writeln(done)@_prolog.
```

The two queries are essentially the same, the first is an F-logic molecule and so it is implemented internally as a tabled XSB predicate. The second query is implemented as a non-tabled predicates. Thus, despite the fact that the two queries are *logically equivalent*, they are not *operationally equivalent* under local scheduling. Indeed, a simple experiment shows that the answers to the above two queries are produced in different orders (as seen by the order of execution of the print statement. In the first query, `?X[foo(?Y)]` is evaluated completely before proceeding to `writeln(done)@_prolog` and thus the executions of `writeln(?X)@_prolog` are grouped together. In the second case, executions of `writeln(?X)@_prolog` and `writeln(done)@_prolog` alternate, because `q/2`is not tabled and thus its evaluation follows the usual Prolog semantics.

On the other hand, if we have

```
?X[foo(?Y)] :- ?X:?Y, writeln(?X)@_prolog.
q(?X,?Y) :- ?X:?Y, writeln(?X)@_prolog.

?- ?X[foo(?Y)], writeln(done)@_prolog.
?- q(?X,?Y), writeln(done)@_prolog.
```

then the two queries will behave the same, as both `q/2` and `?X[foo(?Y)]` would then be implemented internally as tabled predicates. Likewise, if we replace `foo` with `%foo` then the corresponding molecule would be represented internally as a non-tabled predicate. Thus, the two queries in the program

```
?X[%foo(?Y)] :- ?X:?Y, writeln(?X)@_prolog.
%q(?X,?Y) :- ?X:?Y, writeln(?X)@_prolog.

?- %?X[foo(?Y)], writeln(done)@_prolog.
?- %q(?X,?Y), writeln(done)@_prolog.
```

will produce the same result where `a, b, c` and `done` alternate in the output.

17.5 Cuts

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 as far as Database Query Languages are concerned, it is sometimes indispensable when doing "real work", like pretty-printing FLORA-2 programs or implementing a pattern matching algorithm. To facilitate this kind of tasks, FLORA-2 lets the programmer use cuts. However, the current implementation of XSB has a limitation that Prolog cuts cannot "cut across tabled predicates." If you get an error message telling something about cutting across the tables — you know that you 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 or tabled predicate. However, it is OK 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). If you need to use cuts, plan on using procedural methods or non-tabled predicates.

Also, when XSB is configured for *local* scheduling, cuts across tables are much less likely, because under this strategy XSB tries to compute the entire clique of interrelated predicates *before* it proceeds to the next body literal (which could be the dreadful cut). Thus, something like

```
?X[%foo(?Y)] :- ?Z[moo->?W], ?W:?X, !, rest.
```

will not cause problems under the local scheduling, but

```
?X[foo->?Y] :- ?Z[moo->?W], ?W:?X, !, rest.
```

will likely result in a runtime error. The reason is that in the first case the molecule `?X[%foo(?Y)]` is implemented as a non-tabled predicate, so by the time the evaluation reaches the cut, both `?Z[moo->?W]` and `?W:?X` will be evaluated completely and their tables will be marked as "complete." In contrast, in the second example, `?X[foo->?Y]` is implemented as a tabled predicate, which is interrelated with the predicates that are used to implement `?Z[moo->?W]` and `?W:?X`. Thus, the cut would occur in the middle of the computation of the table for `?X[foo->?Y]` and an error will result.

In a future release, XSB will implement a different tabling schema. While cutting across tables will still be prohibited, it will provide an alternative mechanism that achieves many of the goals a cut is used for.

18 Updating the Knowledge Base

*F*LORA-*2* provides primitives to update the runtime database. Unlike Prolog, *F*LORA-*2* does not require the user to define a predicate as dynamic in order to update it. Instead, every predicate and object has a *base part* and a *derived part*. Updates directly change only the base parts and only indirectly the derived parts.

Note that the base part of a predicate or an object contains *both* the facts that were *inserted explicitly* into the database and the facts that you specified in the program. For instance, in

```
p(a).
a[m->b].
```

the fact `p(a)` will be placed in the base part of the predicate p/1 tt and it can be deleted by the `delete` primitive. Likewise, the fact `a[m->b]` is updatable. If you do not want some facts to be updatable, use the following syntax:

```
p(a) :- true.
a[m->b] :- true.
```

*F*LORA-*2* updates can be *non-transactional,* as in Prolog, or *transactional*, as in Transaction Logic [2, 1]. We first describe non-transactional updates.

18.1 Non-transactional (Non-logical) Updates

The effects of non-transactional updates persist even if a subsequent failure causes the system to backtrack.

*F*LORA-*2* supports the following non-transactional update primitives: `insert`, `insertall`, `delete`, `deleteall`, `erase`, `eraseall`. These primitives use special syntax (the curly braces) and are *not* predicates. Thus, it is allowed to have a user-defined predicate such as `insert`.

Insertion. The syntax of an insertion is as follows (note the {,}s!):

insop`{`*literals* `[|` *query*`]}`

where *insop* stands for either `insert` or `insertall`. The *literals* part represents a comma separated list of literals, which can include predicates and F-molecules. The optional part, `|` *query*, is an additional condition that must be satisfied in order for *literals* to be inserted or deleted (depending on what *insop* is). The semantics is that *query* is posed first and, if it is satisfied, *literals* is inserted

(note that the query may affect the variable binding and thus the particular instance of *literals* that will be inserted). For instance, in

```
flora2 ?- insert{p(a),Mary[spouse->Smith,children->Frank]}
flora2 ?- insert{?P[spouse->?S] | ?S[spouse->?P]}
```

the first statement inserts a particular molecule. In the second case, the query `?S[spouse->?P]` is posed and one answer (a binding for `?P` and `?S`) is obtained. If there is no such binding, nothing is inserted and the statement fails. Otherwise, the instance of `?P[spouse->?S]` is inserted for that binding and the statement succeeds.

The insert statement has two forms: `insert` and `insertall` The difference between `insert` and `insertall` is that `insert` inserts only one instance of *literals* that satisfies the formula, while `insertall` inserts *all* instances of the literals that satisfy the formula. In other words, *query* is posed first and *all* answers are obtained. Each answer is a tuple of bindings for some (or all) of the variables that occur in *literals*. To illustrate the difference between `insert` and `insertall`, consider the following queries:

```
flora2 ?- p(?X,?Y), insert{q(?X,?Y,?Z)|r(?Y,?Z)}.
flora2 ?- p(?X,?Y), insertall{q(?X,?Y,?Z)|r(?Y,?Z)}.
```

In the first case, if `p(x,y)` and `r(y,z)` are true, then the fact `q(x,y,z)` is inserted. In the second case, if `p(x,y)` is true, then the update means the following:

> For each `z` such that `r(y,z)` holds, `insert q(x,y,z)`.

The primitive `insertall` is also known as a bulk-insert operator.

Unlike `insert`, the operator `insertall` always succeeds and it always leaves its free variables unbound.

The difference between `insert` and `insertall` is more subtle than it may appear from the above discussion. In the all-answers mode, the above two queries will actually behave the same, because $\mathcal{F}$LORA-*2* will try to find all answers to the query `p(?X,?Y), r(?Y,?Z)` and will do the insertion for each answer. The difference becomes apparent if $\mathcal{F}$LORA-*2* is in one answer at a time mode (because `_one` was executed in a preceding query) or when the all-answers mode is suppressed by a cut as in

```
flora2 ?- p(?X,?Y), insert{q(?X,?Y,?Z)|r(?Y,?Z)}, !.
flora2 ?- p(?X,?Y), insertall{q(?X,?Y,?Z)|r(?Y,?Z)}, !.
```

In such cases, the first query will indeed insert only one fact, while the second will insert all.

Note that literals appearing inside an insert primitive (to the left of the `|` symbol, if it is present) are treated as facts and should follow the syntactic rules for facts and literals in the rule head. In particular, path expressions are not allowed. Similarly, module specifications inside update operators are illegal. However, it is allowed to insert facts into a different module so module specifications are permitted in the literals that appear in the `insert{...}` primitive:

```
flora2 ?- insert{(Mary[children->Frank], John[father->Smith]) @ foomod}
```

The above statement will insert `Mary[children->Frank]` and `John[father->Smith]` into module `foomod`.

Note that module specifications are also allowed in the *condition* part of an update operator (to the right of the `|` mark):

```
flora2 ?- insert{Mary[children->?X]@foobar | adult(?X)@infomod}
```

Updates to Prolog modules is accomplished using the usual Prolog's `assert/retract`:

```
flora2 ?- assert(foo(a,b,c))@_prolog.
```

The following subtleties related to updates of Prolog modules are worth noting. Recall Section 12.4 on the issues concerning the difference between the HiLog representation of terms in $\mathcal{F}$LORA-*2* and the one used in Prolog. The problem is that `foo(a,b,c)` is a HiLog term that Prolog does not understand and will not associate it with the predicate `foo/3` that it might have. To do it right, use explicit conversion:

```
flora2 ?- p2h{?PrologRepr,foo(a,b,c)}, assert(?PrologRepr)@_prolog.
```

This will insert `foo(a,b,c)` into the default XSB module called `usermod`.

If all this looks too complicated, $\mathcal{F}$LORA-*2* provides a higher-level primitive, `@_prologall` (equivalently `@_plgall`), as described in Section 11.7. This module specifier does automatic conversion of terms to and from Prolog representation, so the above example can be written much more simply:

```
flora2 ?- assert(foo(a,b,c))@_prologall.
```

Another possible complication might be that If `foo/3` is defined in another Prolog module, `bar`, and is imported by `usermod`, then the above statement will not do anything useful due to certain idiosyncrasies in the XSB module system. In this case, we have to tell the system that `foo/3` was defined in Prolog module `bar`. Thus, `foo/3` was defined as a dynamic predicate in the module `bar`, we have to write:

```
flora2 ?- assert(foo(a,b,c)@_prolog(bar))@_prolog.
```

Note that if we want to assert a more complex fact, such as `foo(f(a),b,c)`, we would have to use either

```
flora2 ?- assert(foo(f(a)@_prolog(bar),b,c)@_prolog(bar))@_prolog.
```

or `@_prologall`:

```
flora2 ?- assert(foo(f(a),b,c)@_prologall(bar))@_prolog.
```

We should also mention one important difference between insertion of facts in FLORA-2 and Prolog. Prolog treats facts as members of a *list*, so duplicates are allowed and the order matters. In contrast, FLORA-2 treats the database as a *set* of facts with no duplicates. Thus, insertion of a fact that is already in the database has no effect.

Deletion. The syntax of a deletion primitive is as follows:

delop{*literals* [| *query*]}

where *delop* can be `delete`, `deleteall`, `erase`, and `eraseall`. The *literals* part is a comma separated list of F-molecules and predicates. The optional part, |*query*, represents an additional constraint or a restricted quantifier, similarly to the one used in the insertion primitive.

For instance, the following predicate:

```
flora2 ?- deleteall{John[?Year(?Semester)->?Course] | ?Year < 2000}
```

will delete John's course selection history before the year 2000.

Note that the semantics of a `delete`{*literal*|*query*} statement is that first the query *literal* ∧ *query* should be asked. If it succeeds, then deletion is performed. For instance, if the database is

```
p(a). p(b). p(c). q(a). q(c).
```

then the query below:

```
?- deleteall{p(?X)|q(?X)}
```

will succeed with the variable `?X` bound to `a` and `c`, and `p(a)`, `p(c)` will be deleted. However, if the database contains only the facts `p(b)` and `q(c)`, then the above predicate will succeed (`deleteall`always succeeds) and the database will stay unchanged.

FLORA-2 provides four deletion primitives: `delete`, `deleteall`, `erase`, and `eraseall`. The primitive `delete` removes at most one fact at a time from the database. The primitives `deleteall` and `eraseall` are **bulk delete** operations; `erase` is kind of a hybrid: it starts slowly, by deleting one fact, but may go on a joy ride and end up deleting much of your data. These primitives are described below.

1. If there are several bindings or matches for the literals to be deleted, then `delete` will choose only one of them nondeterministically, and delete it. For instance, suppose the database contains the following facts:

   ```
   p(a). p(b). q(a). q(b).
   ```

then

```
?- delete{p(?X),q(?X)}
```

will succeed with `?X` bound to either `a` or `b`, depending on the ordering of facts in the database at runtime.

However, as with insertion, in the all-answers mode the above deletion will take place for each binding that makes the query true. To avoid this, use one answer at a time mode or the cut.

2. In contrast to the plain `delete` primitive, `deleteall` will try to delete all bindings or matches. Namely, for each binding of variables produced by *query* it deletes the corresponding instance of *literal*. If *query* $\wedge$ *literal* is false, the `deleteall` primitive fails. To illustrate, consider the following:

   ```
   flora2 ?- p(?X,?Y), deleteall{q(?X,?Y,?Z)|r(?Y,?Z)}.
   ```

   and suppose `p(x,y)` is true. Then the above statement will, for each `z` such that `r(y,z)` is true, delete `q(x,y,z)`.

   For another example, suppose the database contains the following facts:

   ```
   p(a). q(b). q(c).
   ```

   and the query is `?- deleteall{p(a),q(?X)}`. The effect will be the deletion of `p(a)` and of all the facts in `q`. (If you wanted to delete just one fact in `q`, `delete` should have been used.)

   Unlike the `delete` predicate, `deleteall` *always* succeeds. Also, `deleteall` leaves all variables unbound.

3. `erase` works like `delete`, but with an object-oriented twist: For each F-logic fact, f, that it deletes, `erase` will traverse the object tree by following f's methods and delete all objects reachable in this way. It is a power-tool that can cause maiming and injury. Safety glasses and protective gear are recommended.

   Note that only the base part of the objects can be erased. If the object has a part that is derived from the facts that still exist, this part will not be erased.

4. `eraseall` is the take-no-prisoners version of `erase`. Just like `deleteall`, it first computes *query* and for each binding of variables it deletes the corresponding instance of *literal*. For each deleted object, it then finds all objects it references through its methods and deletes those. This continues recursively until nothing reachable is left. This primitive always succeeds and leaves its free variables unbound.

18.2 Backtrackable (Logical) Updates

The effects of transactional updates are undone upon backtracking, i.e., if some post-condition fails and the system backtracks, a previously inserted item will be removed from the database, and a previously deleted item will be put back.

The syntax of transactional update primitives is similar to that of non-transactional ones and the names are similar, too. The syntax for transactional insertion is:

t_insop{*literals* [| *formula*]}

while the syntax of a transactional deletion is:

t_delop{*literals* [| *query*]}

where `t_insop` stands for either `t_insert` or `t_insertall`, and `t_delop` stands for either of the following four deletion operations: `t_delete`, `t_deleteall`, `t_erase`, and `t_eraseall`. The meanings of *literals* and *query* is the same as in Section 18.1.

`t_insert`, `t_insertall`, `t_delete`, `t_deleteall`, `t_erase`, and `t_eraseall` work similarly to `insert`, `delete`, `deleteall`, `erase`, and `eraseall`, respectively, except that the new operations are transactional. Please refer to Section 18.1 for details of these operations.

To illustrate the difference between transactional and non-transactional updates, consider the following execution trace immediately after the $\mathcal{F}$LORA-*2* system starts:

```
flora2 ?- insert{p(a)}, fail.

No

flora2 ?- p(a).

Yes

flora2 ?- t_insert{q(a)}, fail.

No

flora2 ?- q(a).

No
```

In the above example, when the first `fail` executes, the system backtracks to `insert{p(a)}` and does nothing. Thus the insertion of `p(a)` persists and the following query `p(a)` returns with `Yes`. However, when the second `fail` executes, the system backtracks to `t_insert{q(a)}` and removes `q(a)` that was previously inserted into the database. Thus the next query `q(a)` returns with `No`. This behavior is similar to database transactions, whence the name "transactional" update.

Notes on working with transactional updates. Keep in mind that some things that Prolog programmers routinely do with `assert` and `retract` goes against the very concept of transactional updates.

- `fail`-loops are not going to work (will leave the database unchanged) for obvious reasons. The `while` and `until` loops should be used in such situations.
- Tabled predicates or methods must never depend on transactional updates. First, as explained on page 82, tabled predicates should not depend on any predicates that have side effects, because this rarely makes sense. Second, when evaluating tabled predicates, XSB performs backtracking unbeknownst to the programmer. Therefore, if a tabled predicate depends on a transactional update, backtracking will happen invisibly, and the updates will be undone. Therefore, in such situations transactional updates will have no effect.
- As before, `t_insertall`, `t_deleteall`, and `t_eraseall` primitives always succeed and leave the free variables unbound. Likewise, in the all-answers mode, the primitives `t_insert`, `t_delete`, and `t_erase` behave similarly to the `bt*all` versions in other respects, *i.e.*, they will insert or delete facts for every answer to the associated query. This can be prevented with the use of the cut or the `_one` directive.

Unimplemented: In the current release, arithmetic expressions in the query part of an update must have their variables be bound by the subgoals that precede the update primitive, except that the literal part does not currently bind. For instance,

```
flora2 ?- delete{?X[salary->?Y] | ?Y<20000}.
```

is going to cause a run-time error. This limitation will be removed in a future release.

18.3 Updates and Tabling

Changing tabled predicates or predicates on which tabled predicates depend. We have earlier remarked in Section 17.3 that tabling and database updates do not mix well. One problem is that the results from previous queries are stored in Prolog tables, and database updates do not modify those tables. Thus, in Prolog the user might get the following counterintuitive result, if the predicate `p/1` is tabled:

```
| ?- assert(p(a)).

yes
| ?- p(a)

yes
| ?- retract(p(a)), p(a).

yes
```

What is going on here? The last positive answer is a consequence of the fact that Prolog tables remember that the fact `p(a)` is true from the evaluation of the second query. So, when the same query is asked after `retract`, a "stale" answer is returned from the tables. Similarly, tabling might interact poorly with `assert` in the following case:

```
| ?- p(b).

no
| ?- assert(p(b)), p(b).

no
```

The reason for the bad answer is, again, that Prolog remembers that `p(b)` was false the last time it looked up this fact, even though this answer has become stale after the insertion.

Fortunately, *FLORA-2* is much more update-friendly than plain Prolog, and in situations similar to the above it will behave correctly:

```
flora2 ?- insert{o[m->v]}.

Yes

flora2 ?- o[m->v].

Yes

flora2 ?- delete{o[m->v]}, o[m->v].

No

flora2 ?- insert{o[m->v]}, o[m->v].

Yes
```

Nevertheless, there still are problems with facts that depend through rules on facts that were inserted or deleted. This problem is illustrated by the following program.

```
a[b->c] :- d[e->f].
d[e->f].
```

Consider the following query

```
flora2 ?- a[b->c].

Yes
```

Suppose that next we delete the fact `d[e->f]`. Then we will get the following counterintuitive results:

```
flora2 ?- delete{d[e->f]}.

Yes

flora2 ?- a[b->c].

Yes
```

The reason for this behavior is, as before, that the stale positive answer to the query `a[b->c]` has been recorded in Prolog tables. In order to invalidate this answer it would be necessary to keep track of the dependencies among different facts, which Prolog currently does not do (and it would be very hard and inefficient to keep this information at the $\mathcal{F}$LORA-*2* level).

Nevertheless, $\mathcal{F}$LORA-*2* provides partial solution to this problem in the form of the `refresh{...}` operator, which lets the programmer to explicitly remove stale answers from tables. For instance, in the above case we could do the following:

```
flora2 ?- refresh{a[b->c]}, a[b->c].

No
```

In general, `refresh{...}` can take a comma-separated list of facts to be purged from the tables, and the facts can even contain unbound variables. In the latter case, any stale call that unifies with the given facts will be refreshed. For instance,

```
flora2 ?- refresh{a[b->?X], c:?Y, p(z,?V)@foo}.

Yes
```

will refresh the tables for `a[b->?X]` and `c:?Y` in module `main`, and for `p(z,?V)` in module `foo`.

Sometimes it is desirable to completely get rid of all the information stored in tables (for instance, when it is hard to track down all the facts that might depend on the changed base facts. In such a case, the command

```
flora2 ?- abolish_all_tables.
```

can be used. However, this command is unsafe: If it is executed during the computation of a subquery that involves an F-logic molecule or a tabled predicate, then the system might crash. (This is an unsafe, low level XSB builtin.) The only safe way to execute `abolish_all_tables` is as a *separate* query.

Note: Neither the update operators, such as `insert/delete`, nor the `refresh{...}` operator, nor `abolish_all_tables` can occur under the scope of the negation operator "`not`" (either directly or indirectly). If they do, XSB will likely crash. Typically it will first issue an error message telling you that an update has been issued under the scope of the negation operator.

Tabled predicates that depend on update operations. A related issue is that a tabled predicate (or an F-logic molecule) might occur in the head of a rule that has an update operation in its body, or it may be transitively dependent on such an update. Note that this is different from the previous issue, where tabled predicates did not necessarily depend on update operations but rather on other predicates that were modified by these update operations.

In this case, the update operation will be executed the first time the tabled predicate is evaluated. Subsequent calls will return the predicate truth value from the tables, without invoking the predicate definition. Moreover, if the update statement is non-logical (*i.e.*, non-transactional), then it is hard to predict how many times it will be executed (due to backtracking) before it will start being ignored due to tabling.

If $\mathcal{F}$LORA-*2* compiler detects that a tabled literal depends on an update statement, a warning is issued, because such a dependency is most likely a mistake. This warning is issued also for procedural methods (*i.e.*, Boolean methods of the form `%foo(...)`) when a tabled literal depends on them. Moreover, because non-tabled HiLog predicates are regarded as having procedural side-effect by default, this warning is also issued when a tabled literal depends on non-tabled HiLog predicates.

There are situations, however, when dependency on an update makes perfect sense. For instance, we might be computing a histogram of some function by computing its values at every point and then adding it to the histogram. When a value, $f(a)$, is computed first, the histogram is updated. However, subsequent calls to $f(a)$ (which might be made during the computation of other values for f) should not update the histogram. In this case it makes sense to make $f/1$ into a tabled predicate, whose definition will include an update operator. For this reason, a compiler directive `ignore_depchk` is provided to exempt certain predicates and methods from such dependency checks.

The example below shows the usage of the `ignore_depchk` directive.

```
:- ignore_depchk %ins(?), ?[%?]@?.
t(?X,?Y) :- %ins(?X), ?Y[%close]@_io.
%ins(?X) :- insert{?X}.
```

No dependency warning is issued for this program. However, without the `ignore_depchk` directive, three warnings would be issued saying that tabled literal `t(?X,?Y)` depends on `%ins(?X)`, `?Y[%close]`, and `insert`. Notice that `ignore_depchk %ins(?_)` tells the compiler to ignore not only dependencies on `%ins/1`, but also all dependencies that have `%ins/1` in the path.

The `ignore_depchk` directive can also be used to ignore direct dependencies on updates. For example,

```
:- ignore_depchk insert{?_,?_|?_}.
```

ignores dependencies on conditional insertions which insert two literals such as `insert{a,b,|c,d,e}`. And

```
:- ignore_depchk insert{?}.
```

ignores dependencies on unconditional insertions which insert exactly one literal such as `insert{p(a)}` but not `insert{p(a),p(b)}`.

18.4 Updates and Meta-programming

The update operators can take variables in place of literals to be inserted. For instance,

```
flora2 ?- ?X ~ a[b->c], insert{?X}.
```

One use for this facility is when one module, `foo`, provides methods that allow other modules to perform update operations on objects in `foo`. For instance, `foo` can have a rule

```
%update(?X,?Y) :- delete{?X}, insert{?Y}.
```

Other modules can then issue queries like

```
?- John[salary->?X]@foo, ?Y is ?X+1000,
        %update(John[salary->?X],John[salary->?Y])@foo.
```

18.5 Updates and Negation

Negation applied to methods that have side effects is typically a rich source of trouble and confusion.

First of all, applying negation to $\mathcal{F}$LORA-*2* molecules that involve non-transactional updates does not have logical semantics, and thus the programmer must have good understanding of the *procedural* semantics of $\mathcal{F}$LORA-*2* (Section 17.4). In this case, negation applied to methods or predicates that produce side-effects through updates or I/O is that of negation as failure (Section 13.1).

When only transactional updates are used, the semantics is well defined and is provided by *Transaction Logic* [2, 1]. In particular, negation is also well defined. However, simply negating an update, A is useless in programming, since it simply means to jump to a random state that is not reachable via execution of A. As explained in [2, 1], negation is typically useful only in conjunction with $\wedge$, where it acts as a constraint, or with the hypothetical operator of possibility $\Diamond$. In most cases, when the programmer wants to apply negation to a method that performs logical (transactional) updates, he has $\neg\Diamond$`method` in mind, *i.e.*, a test to verify that execution of `method` is not possible.

At present this operator is not implemented in $\mathcal{F}$LORA-*2* and neither `\+` nor `not` will not produce correct results. The difference with the logical formula $\neg\Diamond$`method` is that the latter does not change the current state of the underlying database, while `\+method` will try to execute `method`. If it succeeds, then `\+method` will fail, but the changes made by `method` will not be undone.

18.6 Words of (Extreme) Caution

XSB has certain well-known bugs, which are very hard to fix and which can corrupt its internal state irreparably. One such thing is the issue of backtracking over updates; especially over deletions.

Here we are not talking about $\mathcal{F}$LORA-*2*'s transactional updates, which are implemented properly and are safe. Instead, a problem arises when backtracking over updates happens in the XSB program into which $\mathcal{F}$LORA-*2* programs are compiled. If this happens, the program might crash or produce random, unexplainable results. Unfortunately, it is not easy to spot such cases by just looking at a source code of your $\mathcal{F}$LORA-*2* program, because much is done by the runtime system behind the scene. Here we provide high-level tips that can help avoid the problem.

First, make sure that when you use *non*-transactional updates and your program backtracks over them, deleted facts are never queried again. This situation is analogous to backtracking over updates in XSB.

Second, $\mathcal{F}$LORA-*2* may backtrack over updates if your top-level query calls an update directly or indirectly. This may happen when the interpreter is in the all-answers mode (`_all`) even if the logic of your program does not call for backtracking. Indeed, in the venerable Prolog way all answers can be obtained only by backtracking, and this is what $\mathcal{F}$LORA-*2* does behind the scene. In this situation, transactional updates are as vulnerable as non-transactional ones because after each iteration that returns an answer to the top level all updates are committed and can no longer be backtracked over (due to the aforesaid XSB bug).

The only (more or less) sure remedy in this situation is to ensure that backtracking does not occur at the top level. For instance, if the top-level query is a procedural method, then semantically it would be correct to put a cut at the end of the query:

```
?- ?X[%some_method(?SomeArgs)], something, !.
```

This will work if XSB is configured for the *local* scheduling strategy (which is currently the default). If it is configured for the *batched* strategy (see XSB manual for how to do this), then the above trick will work only is `something` does not have tabled predicates or those predicates are fully evaluated by the time `something` succeeds.

In some cases cuts can be also inserted in the rules that define procedural methods if the logic of the program ensures that backtracking over those methods should not occur.

19 Insertion and Deletion of Rules

$\mathcal{F}$LORA-*2* supports *non-transactional* insertion of rules into modules as well as deletion of inserted rules. A $\mathcal{F}$LORA-*2* module gets created when a program is loaded into it, as described in Section 2, or it can be created using the primitive `newmodule`. Subsequently, rules can be added to an existing module. Rules that are inserted via the `insertrule` and `_add` commands are called *dynamic* and the rules loaded using the `_load` or `[...]` commands are called *static* or *compiled*. Dynamic rules can be deleted via the `deleterule` command. As mentioned in Section 18, $\mathcal{F}$LORA-*2* predicates and molecules can have both static and dynamic parts and no special declaration is required to

make a predicate dynamic. The same molecule or a predicate can be defined by a mixture of static or dynamic rules.

In this section, we will first look at the syntax of creating new modules. Then we will describe how to insert rules and delete rules. Finally, we address other related issues, including tabling, indexing, and the cut.

19.1 Creation of a New Module and Module Erasure at Run-time

The syntax for creating a new module is as follows:

newmodule{*modulename*}

This creates a blank module with the given name and default semantics. If a module by that name already exists, an error results. A module created using `newmodule` can be used just as any module that was created by loading a user program.

A dual operation to module creation is *erasemodule* with the following syntax:

erasemodule{*modulename*}

19.2 Insertion of Rules

Dynamic rules can be inserted before all static rules, using the primitive `insertrule_a`, or after all static rules, using the primitive `insertrule_z` or just `insertrule`. Several rules can be inserted in the same command. The syntax of inserting a list of rules is as follows:

insruleop{*rulelist*}

where *insruleop* is either `insertrule_a`, `insertrule_z`, or `insertrule`, rulelist is a comma-separated list of rules. The rules being inserted *should not* terminate with a period (unlike the static program rules):

```
?- insertrule_a{?X:student :- %enroll(?X,?_T)}.
```

The above inserts the rule `?X:student :- %enroll(?X,?_T)` in front of the current module.

If a rule is meant to be inserted into a module other than the current one, then the rule needs to be parenthesized *and* the module name must be attached using the usual module operator `@`. If *several* rules need to be inserted using the same command, each rule must be parenthesized. For example, the following statement inserts the same rule into two different modules: the current one and into module `mod1`.

```
?- insertrule_a{(?X:student :- %enroll(?X,?_T)),
                (?X:student :- %enroll(?X,?_T))@mod1}.
```

As a result, the rule `?X:student :- %enroll(?X,?_T)` will be inserted in front of each of these two modules. For this to be executed successfully, the module `mod1` must already exist.

19.3 Deletion of Rules

Rules inserted dynamically using `insertrule_a` can be deleted using the primitive `deleterule_a`, and rules inserted using `insertrule_z` can be deleted using the primitive `deleterule_z`. If the user wishes to delete a rule that was previously inserted using either `insertrule_a` or `insertrule_z` then the primitive `deleterule` can be used. Similarly to rule insertion, several rules can be deleted in the same command:

delruleop{*rulelist*}

where *delruleop* is either `deleterule_a` or `deleterule_z` and *rulelist* is a comma-separated list of rules. Rules in the list mast be enclosed in parentheses and *should not* terminate with a period.

To delete the rules inserted in the second example of Section 19.2, we can use

```
flora2 ?- deleterule_a{(?X:student :- %enroll(?X,?_T)),
                       (?X:student :- %enroll(?X,?_T))@mod1}.
```

or

```
flora2 ?- deleterule{(?X:student :- %enroll(?X,?_T)),
                     (?X:student :- %enroll(?X,?_T))@mod1}.
```

FLORA-*2* provides a flexible way to express rules to be deleted by allowing variable rule head, variable rule body, and variable module specification. For example, rule deletions below are all valid:

```
flora2 ?- deleterule{(?H:-q(?X))@foo}.
flora2 ?- deleterule{(p(?X):-q(?X))@?M}.
flora2 ?- deleterule{?H:-?B}.
```

The last query attempts delete every dynamically inserted rule. So, it should be used with great caution.

We should note that a rule with a composite head, such as

```
o[b->?V1,c->?V2] :- something(?V1,?V2).
```

is treated as a pair of separate rules

```
o[b->?V1] :- something(?V1,?V2).
o[c->?V2] :- something(?V1,?V2).
```

Therefore

```
flora2 ?- deleterule{o[b->?V1] :- something(?V1,?V2)}.
```

will succeed and will delete the first of the above rules. Therefore, the following action will fail afterwards:

```
flora2 ?- deleterule{o[b->?V1,d->?V2] :- ?Body}.
```

A Problem with Cuts What is behind rule insertion is pretty simple. As we know from Section 18, every predicate and object has a base part and a derived part. Now we further divide the derived part into three sub-parts: the *dyna sub-part* (the part that precedes all other facts in the predicate), the static sub-part, and the *dynz sub-part*. All rules inserted using `insertrule_a` go into the dyna sub-part; all the rules in the program file go into the static sub-part; and all the rules inserted using `insertrule_z` go into the dynz sub-part.

This works well when there are no cuts in rules inserted by `insertrule_a`. With the cuts, the program might not behave as expected. For example, if we have the following program:

```
p(?X) :- r(?X).
r(a).
q(b).
?- insertrule_a{p(?X) :- q(?X),!}.
?- p(?X).
```

we normally expect the answer to be `b` only. However, FLORA-*2* will return two answers, `a` and `b`. This is because the cut affects only the dynamic part of p(?X), instead of all the rules for `p/1`.

20 Querying the Rule Base

The rule base can be queried using the primitive `clause`. The syntax of `clause` is as follows:

`clause`{*head,body*}

where *h*ead can be anything that is allowed to appear in a rule head and *b*ody can be anything that can appear in a rule body. In addition, explicit module specifications are allowed in the rule heads in the `clause` primitive. Both *head* and *body* represent templates that unify with the actual rules and those rules that unify with the templates are returned.

The following example illustrates the use of the `clause` primitive. Suppose we have previously inserted several rules:

```
flora2 ?- insertrule_a{tc(?X,?Y) :- e(?X,?Y)}.
flora2 ?- insertrule_a{tc(?X,?Y) :- tc(?X,?Z), e(?Z,?Y)}.
flora2 ?- newmodule{foo}.
flora2 ?- insertrule_a{(tc(?X,?Y) :- e(?X,?Y)@_@)@foo}.
flora2 ?- insertrule_a{(tc(?X,?Y) :- tc(?X,?Z), e(?Z,?Y)@_@)@foo}.
```

Then the query

```
flora2 ?- clause{?X,?Y}.
```

will list all the inserted rules. In this case, four rules will be returned. To query specific rules in a specific module — for example, rules defined for the predicate `tc/2` in the module `foo` — we can use

```
flora2 ?- clause{tc(?X,?Y)@foo,?Z}.
```

We can also query rules by providing patterns for their bodies. For example, the query

```
flora2 ?- clause{?X, e(?_,?_)}.
```

will return the first and the third rules.

Querying the rules with composite involves the following subtlety. Recall from Section 19.3 that a rule with a composite head, such as

```
o[b->?V1,c->?V2] :- something(?V1,?V2).
```

is treated as a pair of rules

```
o[b->?V1] :- something(?V1,?V2).
o[c->?V2] :- something(?V1,?V2).
```

Therefore, if we delete one of these rules, for instance,

```
flora2 ?- deleterule{o[b->?V1] :- something(?V1,?V2)}.
```

then a query with a composite head that involves the head of the deleted rule will fail (unless there is another matching rule). Thus, the following query will fail:

```
flora2 ?- clause{o[b->?V1,d->?V2], Body}.
```

The `clause` primitive can be used to query static rules just as it can be used to query dynamic rules. The normal two-argument primitive queries all rules. If one wants to query only the static (compiled) rules or only dynamic (inserted) rules, then the three-argument primitive can be used. For example,

```
flora2 ?- clause{static,?X,?Y}.
flora2 ?- clause{dynamic,?X,?Y}.
```

Withing the dynamic rules, one can separately query just the dynamic rules that precede all the static rules (using the flag `dyna`) or just those dynamic rules that follow all the static ones (with the `dynz` flag):

```
flora2 ?- clause{dyna,?X,?Y}.
flora2 ?- clause{dynz,?X,?Y}.
```

Due to a limitation of the underlying Prolog system, the `clause` primitive cannot query rules whose size exceeds the limit imposed by the Prolog system. A warning message is issued when a rule exceeds this limit and thus cannot be retrieved by `clause`. The only way to remedy this problem is to split the long rule into smaller rules by introducing intermediate predicates.

21 Aggregate Operations

The syntax for aggregates is similar to the syntax used in the FLORID system.[11] A $\mathcal{F}$LORA-*2* aggregate query has the following form:

`agg{?X[?Gs]` | *query*`}`

where `agg` represents the aggregate operator, `?X` is called the aggregation variable, `?Gs` is a list of comma-separated grouping variables, and *query* is a logical formula that specifies the query conditions. The grouping variables, `?Gs`, are optional. The *query* part can be any combination of conjunction, disjunction, and negation of literals.

All the variables appearing in *query* but not in `?X` or `?Gs` are considered to be existentially quantified. Furthermore, the syntax of an aggregate must satisfy the following conditions:

1. All names of variables in both `?X` and `?Gs` must appear in *query*;
2. `?Gs` should not contain `?X`.

Aggregates are evaluated as follows: First, the query condition specified in *query* is evaluated to obtain all the bindings for the template of the form `<?X, ?Gs>`. Then, these tuples are grouped according to each distinct binding for `<?Gs>`. Finally, for each group, the aggregate operator is applied to the list of bindings for the aggregate variable ?`X`.

The following aggregate operators are supported in $\mathcal{F}$LORA-*2*: `min`, `max`, `count`, `sum`, `avg`, `collectset` and `collectbag`.

The operators `min` and `max` can apply to any list of terms. The order among terms is defined by the Prolog operator `@=<`. In contrast, the operators `sum` and `avg` can take numbers only. If the aggregate variable is instantiated to something other than a 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.

The aggregates `min`, `max`, `sum`, `count`, and `avg` fail if `query` fails. In contrast, `collectbag` and `collectset` succeed even if `query` returns no binding. In this case, these aggregates return the empty list.

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

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):

```
flora2 ?- ?Z = min{?S|John[salary(?Year)->?S]}.
flora2 ?- ?Z = count{?Year|John.salary(?Year) < max{?S|John[salary(?Y)->?S], ?Y < ?Year
flora2 ?- 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, grouping variables 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 query returns, for each employee, a list of years when this employee had salary less than 60. This illustrates the use of the `collectset` aggregate.

```
flora2 ?- ?Z = collectset{?Year[?Who]|?Who[salary(?Year)->?X], ?X < 60}.
?Z = [1990,1991]
?Who = Mary

?Z = [1990,1991,1997]
?Who = John
```

21.1 Aggregation and Set-Valued Methods

Aggregation is often used in conjunction with set-valued methods, and $\mathcal{F}$LORA-*2* provides several shortcuts to facilitate this use. In particular, the operators `->->` and `*->->`, for non-inheritable and inheritable multivalued methods, collects all the values of the given method for a given object in a set. The semantics of these operators is as follows:

```
O[M->->L] :- L=collectset{V|O[M->V]}

O[M*->->L] :- L=collectset{V|O[M*->V]}
```

Note that in `O[M->->L]` and `O[M*->->L]` `L` is a list of oids.

Having special meaning for `->->` and `*->->` means that these constructs *cannot* appear in the head of a rule. One other caveat: recursion through aggregation is not supported and can produce incorrect results.

Sets collected in the above manner often need to be compared to other sets. For this, $\mathcal{F}$LORA-*2* provides another pair of primitives: `+>>` and `*+>>` for non-inheritable and inheritable methods, respectively. The atom of the form `o[m+>>s]` is true if the set of all values of the non-inheritable attribute `m` for object `o` *contains* every element in the list `s`.

For instance, the following query tests whether all Mary's children are also John's children:

```
flora2 ?- Mary[children->->?L], John[children+>>?L].
```

As with `->->` and `*->->`, the use of `+>>` and `*+>>` is limited to rule bodies.

22 Control Flow Statements

*F*LORA-*2* supports a number of control statements that are commonly used in procedural languages. These include `if - then - else` and a number of looping constructs.

22.1 If-Then-Else

This is the usual conditional control flow construct supported by most programming languages. For instance,

```
flora2 ?- if (foo(a),foo2(b)) then (abc(?X),cde(?Y)) else (qpr(?X),rts(??Y)).
```

Here the system first evaluates `foo(a),foo2(b)` and, if true, evaluates `abc(?X),cde(?Y)`. Otherwise, it evaluates `qpr(?X),rts(?Y)`. Note that `if`, `then`, and `else` bind stronger than the conjunction ",", the disjunction ";", etc. This is why the parentheses are needed in the above example.

The abbreviated `if-then` construct is also supported. However, it should be mentioned that *F*LORA-*2* gives a different semantics to `if-then` than Prolog does. In Prolog,

```
..., (Cond -> Action), Statement, ...
```

fails if `Cond` fails and `Statement` is not executed. If the programmer wants such a conditional succeed even if `Cond` fails, then `(Cond->Action; true)` must be used. Our experience shows, however, that it is the latter form that is used in most cases in Prolog programming, so in *F*LORA-*2* the conditional

```
..., if Cond then Action, Statement, ...
```

succeeds even if `Cond` fails and `Statement` is executed next. To fail when `Cond` fails, one should explicitly use `else`: `if Cond then Action else fail`. More precisely:

- `if Cond then Action` fails if and only if `Cond` succeeds but `Action` fails.
- `if Cond then Action else Alternative` succeeds if and only if `Cond` and `Action` both succeed or `Cond` fails while `Alternative` succeeds.

Note that the `if`-statement is friendly to transactional updates in the sense that transactional updates executed as part of an `if`-statement would be undone on backtracking, unless the changes done by such updates are explicitly committed using the `commit` method of the system module `_db` (see Section 29.2).

22.2 Loops

unless-do. This construct is an abbreviation of `if Cond then true else Action`. If `Cond` is true, it succeeds without executing the action. Otherwise, it executes `Action` and succeeds or fails depending on whether `Action` succeeds or fails.

while-do and do-until. These loops are similar to those in C, Java, and the like. In `while Condition do Action`, `Condition` is evaluated before each iteration. If it is true, `Action` is executed. This statement succeeds even if `Condition` fails at the very beginning. The only case when this loop fails is when `Condition` succeeds, but `Action` fails (for all possible instantiations).

The loop `do Action until Condition` is similar, except that `Condition` is evaluated after each iteration. Thus, `Action` is guaranteed to execute at least once.

These loops work by backtracking through `Condition` and terminate when all ways to satisfy it have been exhausted (or when `Action` fails). The loop condition should *not* be modified inside the loop body. If it is modified (*e.g.*, new facts are inserted in a predicate that `Condition` uses), XSB does not guarantee that the changes will be seen during backtracking and thus the result of such a loop is indeterminate. If you need to modify `Condition`, use the statements `while-loop` and `loop-until` described below.

The above loop statements have special semantics for transactional updates. Namely, changes done by these types of updates are *committed* at the end of each iteration. Thus, if `Condition` fails, the changes done by transactional updates that occur in `Cond` are undone. Likewise, if `Action` fails, backtracking occurs and the corresponding updates are undone. However, changes made by transactional update statements during the previous iteration remain committed. If the current iteration finishes then its changes will also remain committed regardless of what happens during the next iteration.

while-loop and loop-until. This pair of loop statements is similar to `while-do` and `do-until`, except that transactional updates are *not* committed after each iteration. Thus, failure of a statement following such a loop can cause all changes made by the execution of the loop to be undone. In addition, `while-loop` and `loop-until` do not work through backtracking. Instead, they execute as long as `Condition` stays true. Therefore, the intended use of these loop statements is that `Action` in the loop body must modify `Condition` and, eventually, make it false (for instance, by deleting objects or tuples from some predicates mentioned in `Condition`).

As in the case of the previous two loops, `while-loop` and `loop-until` succeed even if `Condition` fails right from the outset. The only case when these loops fail is when `Action` fails — see Section 22.3 for ways to avoid this (*i.e.*, to continue executing the loop even when `Action` fails) and the possible pitfalls.

The statements `while-loop` and `loop-until` are more expensive (both time- and space-wise) than `while-do` and `do-until`. Therefore, they should be used only when truly transactional updates are required. In particular, such loops are rarely used with non-transactional updates.

22.3 Subtleties Related to the Semantics of the Loop Statements

Observe that `while-loop` and `loop-until` assume that the condition in the loop is being updated inside the loop body. Therefore, the condition must *not* contain tabled predicates. If such predicates are involved in the loop condition, the loop is likely to execute infinitely many times.

Also, keep in mind that in any of the four loop statements, if `Action` fails before `Condition` does, the loops terminate and fail. Therefore, if the intention is that the loop should continue even if `Action` fails, use the

```
(Action ; true)
```

idiom in the loop body. In case of `while-do` and `do-until`, continuing execution of the loop is not a problem, because these loops work by backtracking through `Condition` and the loop will terminate when there is no more ways to backtrack. However, in case of `while-loop` and `loop-until`, there is a potential pitfall. The problem is that these loops will continue as long as there is a way to satisfy `Condition`. If condition stays true, the loop continues forever. Therefore, the way to use these loops is to make sure that `Condition` is modified by `Action`. If `Action` has non-transactional updates, the user must ensure that if `Action` fails then `Condition` is modified appropriately anyway (for otherwise the loop will never end). If `Action` is fully transactional and it fails, then using the `(Action ; true)` idiom in the loop body will definitely make the loop infinite, so the use of this idiom in the body of `while-loop` and `loop-until` is dangerous if there is a possibility that `Action` will fail, and it is useless if the action is expected to always succeed.

23 Constraint Solving

The following feature temporarily does not work, since beginning with XSB 2.6 constraint solving is being revamped and is not supported.

$\mathcal{F}$LORA-*2* provides an interface to constraint solving capabilities of the underlying Prolog engine. Currently XSB supports linear constraint solving over the domain of real numbers (CLPR). However, we must warn that the XSB implementation of CLPR has many rough spots – do not say that we did not warn! To pass a constraint to a constraint solver in the body of a $\mathcal{F}$LORA-*2* rule (or query), simply include it inside curly braces.

Here is a 2-minute introduction to CLPR. Try the following program:

```
?- insert{p(1.0),p(2.0),p(3.0)}.
?- ?X>0, ?X<5, p(?X).
```

Intuitively, one would expect 2.0 and 3.0 as answers. However, if you actually try to run this program, you will be disappointed — an error message will be reported:

> ++Error[XSB]: [Runtime/P] Type Error: Uninstantiated Arithmetic Expression
> Aborting...

This happens because ordinarily Prolog views `>/2` and `</2` as predicates with infinite number of facts. Since there are infinite number of values for `?X` that make `?X > 0` true, it reasons, the query does not make sense.

Constraint logic programming takes a different view: it considers `?X>0, ?X<5` to be a *constraint* on the set of solutions of the query `p(?X)`. This approach allows Prolog to return meaningful solutions to the above query. However, the user must explicitly tell the system which view to take — the "dumb" view that treats arithmetic built-ins are infinite predicates or a "smart" view, which treats them as constraints. The smart view is indicated by enclosing constraints in curly braces. Thus, the above program becomes:

```
?- [clpr].       % must be loaded prior to the use of constraint solver
?- insert{p(1.0),p(2.0),p(3.0)}.
?- {?X>0, ?X<5}, p(?X).

 ?X = 2.000000e+00

 ?X = 3.000000e+00

 2 solution(s) in 0.0000 seconds
```

Note that the package `clpr` must be loaded in advance.

It should be kept in mind that the constraint solver is very picky about the type of values it is willing to work with. It insists on floats and will refuse to convert integers to floats. For instance, if the insert statement were as follows:

```
flora2 ?- insert{p(1),p(2),p(3)}.
```

then the user would have been rewarded with the following obscure message:

type_error(_h5356 = 3,2,a real number,3)

It is trying to tell the user that a floating number is expected and the integer 3 will not do.

24 Exception Handling

$\mathcal{F}$LORA-*2* supports the common catch/throw paradigm through the primitives `catch{?Goal, ?Error, ?Handler}` and `throw{?Error}`. Here `?Goal` can be any $\mathcal{F}$LORA-*2* query, `?Error` is a HiLog (or Prolog) term, and `?Handler` is a $\mathcal{F}$LORA-*2* query that will be called if an exception that unifies with `?Error` is thrown during the execution of `?Goal`. For instance,

```
%someQuery(?Y) :- ?Y[value->?X], ?X > 0, %doSomethingUseful(?X).
%someQuery(?Y) :- ?Y[value->?X], ?X =< 0, throw{myError('?X non-positive', ?X)}.
```

```
?- %p(?Y), catch{%someQuery(?Y), myError(?Reason,?X), %handleException(?Reason,?X)}.

%handleException(?Reason,?X) :-
        format('~w: ?X=~w~n',[?Reason,?X])@_prolog(format), fail.
```

The `catch` construct first calls the query `%someQuery/1`. If `?X` is positive then nothing special happens, the query executes normally, and `catch{...}` has no effect. However, if `?X` turns out to be non-positive then the query throws an exception `myError('?X non-positive', ?X)`, where `?X` is bound to the non-positive value that was deemed by the logic of the program to be an exceptional situation. The term thrown as an exception is then unified with the term `myError(?Reason,?X)` that was specified in `catch{...}`. If the two terms do not unify (*e.g.*, if the error specified in `catch` was something like `myError(foo,?X)`) then the exception is propagated upwards and if the user program does not catch it, the exception will eventually be caught by the $\mathcal{F}$LORA-*2* command loop. In out concrete case, however, the thrown term and the exception specified in `catch` unify and thus `%handleException/2` is called with `?Reason` and `?X` bound by this unification.

The queries `?Goal` and `?Handler` in the `catch{...}` primitive can be F-logic molecules, not just predicates. However, `?Error` — both in `catch` and in `throw` — must be HiLog or Prolog terms. No molecules are allowed inside these terms unless they are reified. That is, `myError('problem found', a[b->c])` will result in a parser error, but an exception of the form `myError('problem found', ${a[b->c]})` is correct because the molecule is reified.

Some exceptions are thrown by $\mathcal{F}$LORA-*2* itself, and applications might want to catch them:

- `'_$flora_undefined'(?MethodSpec,?ErrMsg)` — thrown when undefinedness checking is in effect (see Section 26.1) and an attempt is made to execute an undefined method or predicate. The first argument in the thrown exception is a specification of the undefined predicate or the method that caused the exception. The second argument is the error message.
- `'_$flora_abort'` or `'_$flora_abort'(?Message)` — thrown when $\mathcal{F}$LORA-*2* encounters other kinds of errors. This exception comes in two flavors: with an error message and without.

  A user program can also throw this exception when immediate exit to the top level is required. The safest way to do so is by calling `%abort(?Message)@_sys`, as explained in Section 29.3.

These exceptions are defined by $\mathcal{F}$LORA-*2* under the symbolic names `FLORA_UNDEFINED_EXCEPTION` and `FLORA_ABORT`. When a user application needs to catch these errors we recommend to include the file `flora_exceptions.flh` in the program and use the above symbolic names. For instance,

```
#include "flora_exceptions.flh"
?- ..., catch{myQuery(?Y),
                FLORA_ABORT(FLORA_UNDEFINED_EXCEPTION(?MethSpec,?Message),?_),
                myHandler(?MethSpec)}.
?- ..., catch{yourQuery(?Y),FLORA_ABORT(?Message,?_),yourHandler(?Message)}.
```

The `catch{...}` primitive can also catch exceptions thrown by the underlying Prolog system. For this to happen you need to know the format (*i.e.*, the exact terms) of the exceptions thrown

by Prolog (which can be found in the manual). One exception that some sophisticated FLORA-2 program might need to catch is thrown by Prolog when a FLORA-2 application calls an undefined Prolog predicate. The format of the corresponding exception term is

```
error(undefined_predicate(PredName,Arity,Module),Message,_)
```

where `Module` is the name of the Prolog module (not FLORA-2 module!) in which the predicate `PredName/Arity` was called. If you call a Prolog predicate in a Prolog module that does not exist, then Prolog will throw the exception

```
error(existence_error(module,Module),_)
```

Note that the thrown exception will contain a module name but not the predicate.

Although Prolog (obviously) throws Prolog terms as exceptions, there is no need to worry about making sure that the terms caught by the FLORA-2`catch{...}` primitive are also specified as Prolog terms. he primitive takes care of the Prolog-to-HiLog conversion automatically.

25 Primitive Data Types

An extensive data type support is being planned for FLORA-2 in the future. At present, FLORA-2 supports the built-in data types `_long`, *_integer*, `_double`, *_decimal*, *_string*, `_symbol`, `_object`, `_iri`, (international resource identifier), `_time`, `_date`, `_dateTime`, and `_duration`.

Following the now accepted practice on the Semantic Web, FLORA-2 denotes the constants that belong to a particular primitive data type using the idiom `"`*literal*`"^^`*type*. The *literal* part represents the value of the constant and the *type* part is the type. For instance, `"2004-12-24"^^_date`, `"2004-12-24T15:33:44"^^_dateTime`.

A type name must be an atom. Some data types, like time, dateTime, etc., are exact analogues of the corresponding XML Schema types. In this case, their names will be denoted using symbols that have the form of a URI. For instance, `'http://www.w3.org/2001/XMLSchema#time'`. However, for convenience, all type names will have one or more FLORA-2-specific abbreviated forms, such as `_time` or `_t`. These abbreviated forms are case-insensitive. So, `_time` and `_TiMe` are assumed to be equivalent. In addition, when the type names have the form of an IRI, the compact prefix representation is supported (see Section 25.2 below). For instance, if `xsd` is a prefix name for `'http://www.w3.org/2001/XMLSchema#'` then the constant `"12:33:55"^^'http://www.w3.org/2001/XMLSchema#time'` can be written as `"12:33:55"^^xsd#time'`. Taking into the account the various abbreviations for this data type, we can also write it as `"12:33:55"^^_time` or even `"12:33:55"^^_t`.

Variables can be also *typed*, i.e., restricted to be bound only to objects of a particular primitive data type. The notation is `?`*variablename*`^^`*typename*. For instance, the variable `?X^^_time` can be bound only to constants that have the primitive types `_time`.

The methods that are applicable to each particular primitive type vary from type to type. However, certain methods are more or less common:

- _toString, which applies to a data type constant and returns its printable representation. For instance, if ?Y is bound to "12:44:23"^^_time then ?Y[_toString->'12:44:23'] will be true.
- _toType(parameters), which applies to the any class corresponding to a primitive data type (for instance, _time). Most types will have two versions of this method. One will apply to arguments that represent the components of a data type. For instance, _time[_toType(12,23,45) -> "12:23:45"^^_time]. The other will apply to the constant symbol representation of the data type. For instance, _time[_toType('12:23:45')->"12:23:45"^^_time].
- _isTypeOf(constant), which applies to every data type class (e.g., _time) and determines whether constant has the given primitive type (_time in this example).
- _equal(constant), which tells when the given datatype constant equals some other term.
- _lessThan(constant), which tells when one constant is less than some other terms. For integers, floats, time, dates, durations, and strings, this method corresponds to the natural order on these types. For other types this method returns false.
- _typeName, which tells the type name (and thus also class) of the given data type.

All these methods are available in *FLORA-2* system module _basetype.

In addition, each primitive data type has a builtin class associated with it. For instance, the primitive data type _integer has an associated class named _integer and the data type _dateTime has an associated class under the same name.

Note: Since builtin classes have infinite extensions, you can only have ground membership tests with respect to these classes. Non-ground tests are permitted, but are evaluated to **false**. For instance, the following query fails.

```
flora2 ?- ?X:\_symbol.

No
```

We now describe each data type separately.

25.1 *FLORA-2* Symbols

Before describing the actual data types, we would like to remind that in Section 5.1 we introduced alphanumeric constants, such as abc12, and sequences of symbols enclosed in single quotes, such as 'aaa 2*)@', and called them *constant symbols*. These are not the only constants in *FLORA-2*. In the following subsections we will introduce typed literals that represent time, URIs, and more.

Constant symbols belong to *FLORA-2* builtin class _symbol. In addition, this class contains IRIs, which are described next. An IRI of the form "some-string"^^_iri is assumed to be identical to a constant symbol 'some-string'. However, this feature has not been fully implemented. Namely, a query

```
?- foobar = "foobar"^^_iri.
```

fails. However, the methods defined for the IRIs are applicable. For instance,

```
?- 'http://foo@bar.com'[_scheme->?X]@_basetype.
```

binds `?X` to `http`, as is expected of URIs. Such methods will give syntax error if applied to symbols that are not IRIs:

```
?- foo[_host->?X]@_basetype.

++Abort[FLORA]> invalid IRI literal
```

25.2 The _iri Data Type

The canonical representation of the constants of type IRI (international resource identifiers, a generalization of IRIs, universal resource identifiers) is `"`*`some iri`*`"^^_iri`, where *literal* must have a lexical form corresponding to IRIs on the WWW. IRIs have shorthand notation `_"`*`some iri`*`"`, as mentioned before. The full IRI name of this type is `'http://www.w3.org/2007/rif#iri'`.

IRIs can come in the usual full form or in an abbreviated form known as the *ciri* form (for *compact IRI*).

An a *compact form* of an IRI (ciri) consists of a prefix and a local-name as follows: *PREFIXNAME#LOCALNAME*. Here *PREFIXNAME* is an alphanumeric identifier that must be defined as a shortcut for an IRI elsewhere (see below). *LOCALNAME* can be a string, an alphanumeric identifier, or a quoted atom. (If *LOCALNAME* contains non-alphanumeric symbols, it must be enclosed in double quotes as in `"ab%20"`.) A compact IRI is treated as a macro that expands into a full IRI by concatenating the expansion of *PREFIXNAME* with LOCALNAME.

The prefix of a compact IRI must be defined as follows:

```
:- iriprefix PREFIXNAME = PREFIXIRI.
```

Here `PREFIXIRI` can be an alphanumeric identifier, a quoted atom, or a character list. Prefixes can also be defined on command line at run time:

```
?- iriprefix PREFIXNAME = PREFIXIRI.
```

Such a prefix becomes defined only after the command is executed. If a prefix is used before it is defined, an error will result. For example,

```
:- iriprefix w3c = "http://www.w3c.org/", AAAWEB = "http://www.AAA.com/".
```

Defines two prefixes, which can be used in subsequent commands like this:

```
?- ?X = w3c#a.
```

This will bind `?X` to `_"http://www.w3c.org/a"`. Likewise,

```
?- ?Y = AAAWEB#"ab%20"
```

binds `?Y` to `_"http://www.AAA.com/ab%20"`.

Note that prefix definitions are local to the module where they are defined. If we define the following prefixes in module `foo`:

```
?- iriprefix W3="http://www.w3.org/", W4="w4/".
```

and then load the following file into module `main`

```
:- iriprefix W3 = "http://w3.org/".
C[a->_"http://www.w3.org/abc"].
D[a->_"http://w3.org/cde"].
r(?X):-?X[a->(W3#abc)@foo].
s(?X):-?X[a->W3#cde].
```

then the different occurrences of `W3` will have different expansions. Thus, the answer to

```
?- r(?X).
```

will be `C` and the answer to

```
?- s(?X).
```

will be `D`. Note that a reference to `W3#...` in a module where the prefix `W3` is not defined will result in an error.

For convenience, some IRI prefixes are predefined:

```
xsd   'http://www.w3.org/2001/XMLSchema#'
rdf   'http://www.w3.org/1999/02/22-rdf-syntax-ns#'
rdfs  'http://www.w3.org/2000/01/rdf-schema#'
owl   'http://www.w3.org/2002/07/owl#'
rif   'http://www.w3.org/2007/rif#'
```

However, one can always override these builtin definitions using either a compile time directive `iriprefix` or a runtime query `iriprefix`.

All constants of the primitive type IRI are members of the built-in class `_iri`.

The IRI data type supports the following methods, which are available in *FLORA-2* module `_basetype`. They are described here by their signatures.

Class methods:

- _iri[_toType(_symbol) => _iri]
- _iri[=> _isTypeOf(_object)]

Component methods:

- _iri[_scheme *=> _symbol]
- _iri[_user *=> _symbol]
- _iri[_host *=> _symbol]
- _iri[_port *=> _symbol]
- _iri[_path *=> _symbol]
- _iri[_query *=> _symbol]
- _iri[_fragment *=> _symbol]

Note that the exact meaning of the above components depends on the URI scheme. For `http`, `ftp`, `file`, etc., the meaning the first five components is clear. The query is an optional part of the IRI that follows the ?-sign, and fragment is the last part that follows #. Some components might be optional for some URI schemes. For instance, for the `urn` and `file` schemata, only the path component is defined. For `mailto` scheme, port, path, query, and fragment are not defined. If a scheme is not recognized then the part of the URI that follows the scheme goes into the `path` component unparsed.

Other methods:

- _iri[_toString *=> _symbol]
- _iri[*=> _equals(_object)]
- _iri[_typeName *=> _symbol]

Examples:

- _"http://foo.bar.com/abc"
- "http://foo.bar.com/abc"^^_iri
- ?- _iri[_toType('http://foo.bar.com/abc') ->
  "http://foo.bar.com/abc"^^_iri]@_basetype
- ?- _"http://foo.bar.com/abc"^^_iri[_host -> 'foo.bar.com']@_basetype

25.3 The Primitive Type _dateTime

This data type corresponds to the XML Schema `dateTime` type. The constants of this data type have the form `"ZYYYY-MM-DDTHH:MM:SS.sZHH:MM"^^_dateTime`. The symbols `-`, `:`, `T`, and `.` are part of the syntax. The leftmost `Z` is an optional sign (`-`). The part that starts with the second `Z` is optional and represents the time zone (the second `Z` is a sign, which can be either `+` or `-`; note that the first `Z` can be only the minus sign or nothing). The part that starts with `T` is also optional; it represents the time of the specified day. The part of the time component of the form `.s` represents fractions of the second. Here `s` can be any positive integer.

The constants of this primitive type all belong to the class `_dateTime`. The name of this type has the following synonyms: `_dt`, `'http://www.w3.org/2001/XMLSchema#dateTime'`.

The following methods are available in the *FLORA-2* system module `_basetype`; they are described by their signatures below.

Class methods:

- `_dateTime[_toType(_integer,_integer,_integer,_integer,_integer,_integer,_decimal,_integer,_integer,_integer) => _dateTime]`
  The meaning of the arguments is as follows (in that order): date sign, year, month, day, hour, minute, second, zone sign, zone hour, zone minute. All arguments, except date sign and zone sign, are assumed to be positive integers, while date sign and zone sign can be either 1 or -1.
- `_dateTime[_toType(_symbol) => _dateTime]`
- `_dateTime[=> _isTypeOf(_object)]`
  Tells if object belongs to the primitive type `_dateTime`.

Component methods:

- `_dateTime[_dateSign *=> _integer]`
- `_dateTime[_year *=> _integer]`
- `_dateTime[_month *=> _integer]`
- `_dateTime[_day *=> _integer]`
- `_dateTime[_hour *=> _integer]`
- `_dateTime[_minute *=> _integer]`
- `_dateTime[_second *=> _integer]`
- `_dateTime[_zoneSign *=> _integer]`
- `_dateTime[_zoneHour *=> _integer]`
- `_dateTime[_zoneMinute *=> _integer]`

Other methods:

- _dateTime[_toString *=> _symbol]
- _dateTime[*=> _equals(_object)]
- _dateTime[*=> _lessThan(_object)]
- _dateTime[_typeName *=> _symbol]
- _dateTime[_add(_duration) *=> _dateTime]

Examples:

- "2001-11-23T12:33:55.123-02:30"^^_dateTime
- "2001-11-23T12:33:55.123-02:30"^^'http://www.w3.org/2001/XMLSchema#dateTime'
- "2001-11-23"^^_dateTime
- "-0237-11-23T12:33:55"^^_dateTime
  Note that this date refers to year 238 BCE.
- ?- "2001-11-23"^^_dateTime[_day -> 23]@_basetype
- ?- "2001-11-23"^^_dateTime[_toString -> '2001-11-23T00:00:00+00:00']@_basetype
- ?- "2001-11-23T18:33:44-02:30"^^_dateTime[_add("-P22Y2M10DT1H2M3S"^^_duration)
  -> "1979-09-13T17:31:41-02:30"^^_dateTime]@_basetype

25.4 The Primitive Type _date

This type corresponds to the XML Schema `date` type. Constants of this type have the form `"ZYYYY-MM-DDSHH:MM"^^_date`. The symbols `-` and `:` are part of the syntax. The symbol `S` represents the timezone sign (`+` or `-`). The timezone part (beginning with `S`) is optional. The leftmost `Z` is the optional sign (`-`). Note that unlike `_dateTime`, which represents a single time point, `_date` represents *duration* of a single day.

All constants of this type belong to the built-in class `_date`. The type name `_date` has the following synonyms: `_d`, `'http://www.w3.org/2001/XMLSchema#date'`.

The following methods are defined for this type and are available through the system module `_basetype`.

Class methods:

- `_date[_toType(_integer,_integer,_integer,_integer,_integer,_integer,_integer) => _date]`
  The meaning of the arguments is as follows (in that order): date sign, year, month, day, zone sign, zone hour, zone minute. All arguments, except date sign and zone sign, are assumed to be positive integers, while date sign and zone sign can be either 1 or -1.
- `_date[_toType(_symbol) => _date]`
- `_date[=> _isTypeOf(_object)]`
  Tells if object belongs to the primitive type `_date`.

Component methods:

- `_date[_dateSign *=> _integer]`
- `_date[_year *=> _integer]`
- `_date[_month *=> _integer]`
- `_date[_day *=> _integer]`
- `_date[_zoneSign *=> _integer]`
- `_date[_zoneHour *=> _integer]`
- `_date[_zoneMinute *=> _integer]`

Other methods:

- `_date[_toString *=> _symbol]`
- `_date[*=> _equals(_object)]`
- `_date[*=> _lessThan(_object)]`
- `_date[_typeName *=> _symbol]`
- `_date[_add(_duration) *=> _date]`

Examples:

- `"2001-11-23-2:30"^^_date`
- `"2001-11-23"^^_date`
- `"-237-11-23"^^_date`
  Note that this date refers to year 238 BCE.

- `?- "2001-11-23"^^_date[_day -> 23]@_basetype`
- `?- "2001-11-23"^^_date[_toString -> '2001-11-23+00:00']@_basetype`
- `?- "2001-11-23-02:30"^^_date[_add("-P2Y2M10D"^^_duration) -> "1979-09-13-02:30"^^_dt]@_basetype`.
  Note that when adding duration to a date, the time-part of the duration constant must be empty.

25.5 The Primitive Type _time

This primitive type corresponds to the XML Schema `time` data type Constants of this type have the form `"HH:MM:SS.sZHH:MM"^^_time`. The symbols `:` and `.` are part of the syntax. The part `.s` is optional. It represents fractions of a second. Here `s` can be any positive integer. The sign `Z` represents the sign of the timezone (`+` or `-`). The following `HH` represents time zone hours and `MM` time zone minutes. The time zone part is optional.

The name of this type has the following alternative versions: `_t` and `'http://www.w3.org/2001/XMLSchema#time'`. All constants of that type are also assumed to be members of the built-in class `_time`.

The following methods are available for the class `_time` and are provided by the module `_basetype`. Their signatures are given below.

Class methods:

- `_time[_toType(_integer,_integer,_decimal,_integer,_integer,_integer) => _time]`
  The arguments represent hour, minute, second, time zone sign, time zone hour, and time zone minute.
- `_time[_toType(_symbol) => _time]`
- `_time[=> _isTypeOf(_object)]`
  Tells if object belongs to the primitive type `_time`

Component methods:

- `_time[_hour *=> _integer]`
- `_time[_minute *=> _integer]`
- `_time[_second *=> _integer]`
- `_time[_zoneSign *=> _integer]`
- `_time[_zoneHour *=> _integer]`
- `_time[_zoneMinute *=> _integer]`

Other methods:

- `_time[_toString *=> _symbol]`
- `_time[*=> _equals(_object)]`
- `_time[*=> _lessThan(_object)]`
- `_time[_typeName *=> _symbol]`
- `_time[_add(_duration) *=> _time]`

Examples:

- `"11:24:22"^^_time`
- `"11:24:22"^^'http://www.w3.org/2001/XMLSchema#time'`
- `?- _time[_toType(12,44,55) -> "12:44:55"^^_time]@_basetype`
- `?- "12:44:55"^^_time[_minute -> 44]@_basetype`
- `?- "12:44:55"^^_time[_toString -> '12:44:55']@_basetype`
- `?- "12:44:55"^^_time[_add("PT2M3S"^^_duration) -> "12:46:58"^^_time]@_basetype`
  Note that when adding duration to time, the date-part of the duration constant must *not* be present.

25.6 The Primitive Type _duration

The primitive type duration corresponds to the XML Schema `duration` data type. The constants that belong to this type have the form `"sPnYnMnDTnHnMnS"^^_duration`. Here `s` is optional sign `-`, `P`indicates that this is a duration data type, and `Y`, `M`, `D`, `H`, `M`, `S` denote year, month, date, hour, minutes, and seconds. `T` separates date from time. The symbols `P`, `Y`, `M`, `D`, `H`, `M`, and `S` are part of the syntax. The symbol `n` stands for any positive number (for instance, the number of hours can be more than 12 and the number of minutes and seconds can exceed 60). The part that starts with `T` is optional and any element in the date and the time parts can be omitted.

The constants of this data type all belong to the class `_duration`.

The type name has the following synonyms: `'http://www.w3.org/2001/XMLSchema#duration'`, `_du`.

The following classes are available in module `_basetype`. Their signatures are shown below.

Class methods:

- _duration[_toType(_integer,_integer,_integer,_integer,_integer,
  => _duration]
  The meaning of the arguments (in that order) is: year, month, day, hour, minute, second.
- _duration[_toType(_symbol) => _duration]
- _duration[=> _isTypeOf(_object)]
  Tells if am object belongs to the primitive type _duration.

Component methods:

- _duration[_year *=> _integer]
- _duration[_month *=> _integer]
- _duration[_day *=> _integer]
- _duration[_hour *=> _integer]
- _duration[_minute *=> _integer]
- _duration[_second *=> _integer]

Other methods:

- _duration[_toString *=> _symbol]
- _duration[*=> _equals(_object)]
- _duration[*=> _lessThan(_object)]
- _duration[_typeName *=> _symbol]
- _duration[_add(_duration) *=> _duration]

Examples:

- "P5Y5M10DT11H24M22S"^^_duration
- ?- "-P2Y05M10DT11H24M22"^^_duration[_minute -> 24]@_basetype

25.7 The Primitive Type _boolean

This corresponds to the XML Schema Boolean type. Constants of this type have the form `"true"^^_boolean "false"^^_boolean` or the shorter form `true`, `false`. A synonym for this type name is `'http://www.w3.org/2001/XMLSchema#boolean'`.

All constants in this type belong to the built-in class `_boolean`. The following methods are available in module `_basetype`.

Class methods:

- `_boolean[_toString => _symbol]`
- `_boolean[=> _isTypeOf(_object)]`

Other methods:

- `_boolean[_toString *=> _symbol]`
- `_boolean[*=> _equals(_object)]`
- `_boolean[*=> _lessThan(_object)]`
  Note: `_false[_lessThan(_true)]`.
- `_boolean[_typeName *=> _symbol]`
- `_boolean[_rawValue *=> _symbol]`
  Extract the short representation value from the `_boolean` data type.

Caveat: The long form and the short form must really be the same, i.e., `"true"^^_boolean` and `true` must denote the same constant. However, this has not been implemented yet. To extract the short representation part from a _boolean data type one should use the method `_rawValue`. For instance, `?- "true"^^_boolean[_rawValue->?X]@_basetype`.

25.8 The Primitive Type _double

This corresponds to the XML Schema type `double`. The constants in this type all belong to the class `_double` and have the form `"value"^^_double`, where `value` is a floating point number that uses the regular decimal point representation with an optional exponent. Doubles have a short form where the "..."^^_double wrapper is removed.

This type name has a synonym `'http://www.w3.org/2001/XMLSchema#double'`. The following methods are available for type `_double` in module `_basetype`.

Class methods:

- `_double[_toType(_decimal) => _double]`
  Converts decimals to doubles. Error, if overflow.
- `_double[_toType(_long) => _double]`
  Converts long integers to doubles.
- `_double[=> _isTypeOf(_object)]`

Instance methods:

- `_double[_floor *=> _integer]`
- `_double[_ceiling *=> _integer]`
- `_double[_round *=> _integer]`

Other methods:

- `_double[_toString *=> _symbol]`
- `_double[*=> _equals(_object)]`
- `_double[*=> _lessThan(_object)]`
- `_double[_typeName *=> _symbol]`
- `_double[_rawValue *=> _number]`
  Extract the number part of the `_double` data type.

Examples: `"2.50"^^_double`, `2.50`, `25E-1`.

Caveat: The long form and the short form must really be the same, i.e., `"2.50"^^_double` and `2.50` must denote the same constant. However, this has not been implemented yet. In fact, it is not even possible to do arithmetics with the long representation of doubles. To extract the number part from a _double data type one should use the method `_rawValue`. For instance, `?- "1.2"^^_double[_rawValue->?X]@_basetype`.

25.9 The Primitive Type _long

This data type corresponds to XML Schema's long integers. The constants in this data type belong to class `_long` and have the form `"value"^^_long`, where value is an integer in its regular representation in the decimal system. A shorter form without the `"..."^^_long` wrapper is also allowed. This type name has a synonym: `'http://www.w3.org/2001/XMLSchema#long'`.

Class methods:

- `_long[_toType(_symbol) => _long]`
  Converts strings to long integers, if the string represents an integer in textual form. If it does not then this method fails.
- `_long[_toType(_integer) => _long]`
  Converts long integers to arbitrary big integers.
- `_integer[=> _isTypeOf(_object)]`

Other methods:

- `_long[_toString *=> _symbol]`
- `_long[*=> _equals(_object)]`
- `_long[*=> _lessThan(_object)]`
- `_long[_typeName *=> _symbol]`
- `_long[_rawValue *=> _number]`
  Extract the number part of the `_long` data type.

Examples: `123`, `55`, `"55"^^_long`.

Caveat: The long form and the short form must really be the same, i.e., `"123"^^_long` and `123` must denote the same constant. However, this has not been implemented yet. In fact, it is not even possible to do arithmetics with the long representation of long integers. To extract the number part from a _long data type one should use the method `_rawValue`. For instance, `?- "12"^^_long[_rawValue->?X]@_basetype`.

25.10 The Primitive Types _decimal and _integer

At present, *FLORA-2* does not implement the `_decimal` and the `_integer` types, which correspond to XML Schema arbitrary precision types decimal and integer. Instead, `_decimal` is a synonym for `_double` and `_integer` for `_long`. As usual, there are corresponding classes `_integer` and `_decimal`.

25.11 The Primitive Type _string

This corresponds to the XML Schema type string. The constants in this class belong to type `_string` and the type name has the synonym `http://www.w3.org/2001/XMLSchema#string`. The values of this class have the form `"value"^^_string`. Alphanumeric strings that start with a letter do not need to be quoted. In the full representation (with the `"..."^^_string` wrapper), the

double quote symbol and the backslash must be escaped with a backslash. In short representation, the single quote symbol and the backslash must be escaped with a backslash.

The following methods are available in module `_basetype`:

Class methods:

- `_string[=> _isTypeOf(_object)]`
- `_string[_toType(_symbol) => _string]`

Instance methods:

- `_string[*=> _contains(_string)]`
- `_string[_concat(_string) *=> _string]`
- `_string[_reverse *=> _string]`
- `_string[_length *=> _integer]`
- `_string[_toUpper *=> _string]`
- `_string[_toLower *=> _string]`
- `_string[*=> _startsWith(_string)]`
- `_string[*=> _endsWith(_string)]`
- `_string[_substring(_integer,_integer) *=> _string]`
  Returns a substring of the object string, where the starting and the ending position of the substring are given by the arguments of the method. -1 in argument 2 means the end of the string.

Other methods:

- `_string[*=> _equals(_object)]`
- `_string[*=> _lessThan(_object)]`
- `_string[_typeName *=> _symbol]`

Examples:

- `"abc"^^_string`
- `'abc'`
- "a `string\n"^^_string`
- 'a `string\n'`
- `'a\tstring\b'`
- `'string with a \'quoted\' substring'`

Caveat: The long form and the short form must really be the same, i.e., `"abc"^^_string` and `abc` must denote the same constant. However, this has not been implemented yet. To extract the atom part from a _long data type one should use the method `_rawValue`. For instance, `?- "abc"^^_string[_rawValue->?X]@_basetype`.

25.12 The Primitive Type _list

This is the usual Prolog list type. The members of this type have the form `[elt1, ..., eltn]^^_list` (short form `[elt1, ..., eltn]`) and belong to class `_list`.

The following methods are available from the standard module `_basetype`:

Class methods:

- `_list[=> _isTypeOf(_object)]`
- `_list[_toType(_list) => _list]`

Other methods:

- `_list[*=> _contains(_list)]`
  Tells if a list object contains the method's argument as a sublist.
- `_list[*=> _member(_object)]`
  The method's argument and the list-object may not be fully ground. In this case, the method succeeds, if the argument to the method unifies with a member of the list.
- `_list[_append(_list) *=> _list]`
- `_list[_length *=> _long]`
  Computes the length of the list.
- `_list[_reverse *=> _list]`

- `_list[_sort *=> _list]`
- `_list[*=> _startsWith(_list)]`
- `_list[*=> _endsWith(_list)]`
- `_list[*=> _subset(_list)]`
  True if the list object contains the argument list.

Other methods:

- `_list[_toString *=> _symbol]`
- `_list[*=> _equals(_object)]`
- `_list[_typeName *=> _symbol]`

Examples:

- `[a,b,c]`
- `[a,b|?X]`
- `[a,b,c|[d,e]]`

26 Debugging User Programs

$\mathcal{F}$LORA-*2* comes with an interactive, Prolog-style debugger, which is described in Appendix B. The compiler makes many useful checks, such as the occurrence of singleton variables, which is often an error (see Section 5.1). More checks will be provided in the future.

In addition, it is possible to tell $\mathcal{F}$LORA-*2* to perform various run-time checks, as described below.

26.1 Checking for Undefined Methods and Predicates

$\mathcal{F}$LORA-*2* has support for checking the invocation of undefined methods and predicates at run time. This feature can be of great help because a trivial typo can cause a method/predicate call to fail, sending the programmer on a wild goose chase after a hard-to-find bug. It should be noted, however, that enabling these checks can slow the runtime by up to 2 times (typically about 50% though), so we recommend this to be done during debugging only.

To enable runtime checks for undefined invocations, $\mathcal{F}$LORA-*2* provides two methods, which can be called at any time during program execution (and thus enable and disable the checks dynamically):

```
?- Method[%_mustDefine(?Flag)]@_sys.
?- Method[%_mustDefine(?Flag(?Module))]@_sys.
```

The argument `?Flag` can be `on`, `off`, or it can be a variable. The argument `?Module` must be a valid loaded FLORA-*2* module name or it can be a variable. When the flag argument is `on`, the first method turns on the checks for undefinedness in all modules. The second method does it in a specific module. When the flag argument is off, the above methods turn the undefinedness checks off globally or in a specific module, respectively.

When either `?Flag` or `?Module` (or both) is a variable, the above methods do not change the way undefined calls are treated. Instead, they query the state of the system. For instance, in

```
?- Method[%_mustDefine(?Flag)]@_sys.
?- Method[%_mustDefine(?Flag(foo))]@_sys.
?- Method[%_mustDefine(on(?Module))]@_sys.
```

the first query binds `?Flag` to `on` or `off` depending on whether the checks are turned on or off globally. The second query reports on the state of the undefinedness checks in FLORA-*2* module `foo`, while the third query tells in which modules these checks are turned on.

In addition to turning on/off the checks for undefinedness on the per-module basis, FLORA-*2* provides a way to turn off such checks for individual predicates and methods:

```
?- Method[%_mustDefine(off,Predicate/Method-spec)]@_sys.
```

For example,

```
?- Method[%_mustDefine(off,?(?)@foo)]@_sys.
```

specifies that all undefinedness errors of predicates that unify with `?(?)@foo` are ignored, provided that `foo` is a loaded module. Note that the module must *always* be specified. For instance, to ignore undefinedness checking in the current module, use

```
?- Method[%_mustDefine(off,?(?)@ _@)]@_sys.
```

Note that the use of the current module symbol `_@` is essential in this example. Omitting it is probably not what you want because the module specification `_sys` propagates inward and so the above statement (without the `_@`) would turn off undefinedness checks in module `_sys` instead of the current module.

One can also turn undefinedness checks is *all* modules by putting a variable in the module position:

```
?- Method[%_mustDefine(off,?(?)@ ?Mod)]@_sys.
```

However, this must *not* be an anonymous variable like `?`, `?_`, or a don't care variable like `?_Something`. If one uses an anonymous or a don't care variable then undefinedness checks will be ignored only in some randomly picked module.

A pair of parenthesis is needed when multiple predicates/methods are listed in one call.

```
?- Method[%_mustDefine(off,(?:class@foo, ?[%?]@ _@))]@_sys.
```

The undefinedness exception in $\mathcal{F}$LORA-*2* can be caught using $\mathcal{F}$LORA-*2*'s `catch{...}` builtin. For instance, suppose `FOO` is a predicate or an F-molecule whose execution might trigger the undefinedness exception. Then we can catch this exception as follows:

```
#include "flora_exceptions.flh"

  ..., catch{FOO, FLORA_UNDEFINED_EXCEPTION(?Call,?ErrorMessage), handler(?Call)}, ...
```

Here `FLORA_UNDEFINED_EXCEPTION` is the exception name defined in the $\mathcal{F}$LORA-*2* system file `flora_exceptions.flh`, which must be included as shown. The predicate `handler/1` is user-defined (can be a molecule as well), which will be called when an undefinedness exception occurs. The variable `?Call` will be bound to an internal representation of the method or predicate call that caused the exception. For instance, if we define

```
handler(?_) :- !.
```

then the undefinedness exception that occurs while executing *FOO* will be ignored and the call to *FOO* will succeed.

Undefinedness check and meta-programming. We should note one subtle interaction between these checks and meta-programming. Suppose your program does not have any class membership facts and the undefinedness checks are turned on. Then the meta-query

```
flora2 ?- a:?X.
```

would cause the following error:

++Error[FLORA]: Undefined class ?:? in user module main

Likewise, if the program does not have any method definitions, the query `?- ?X[?Y->?Z].` would cause an error. This might not be what one expects because the program in question might be *exploring* the schema or the available data, and the intention in the above cases might be to fail rather than to get an error.

One way of circumventing this problem is to insert some unusual facts into the database and special-case them in the program. For instance, you could put the following facts into the program to silence the above errors:

```
ads_asd_fsffdfd : ads_asd_fsffdfd.
ads_asd_fsffdfd[ads_asd_fsffdfd -> ads_asd_fsffdfd].
```

You can then arrange the logic of your program so that anything that contains `ads_asd_fsffdfd` is discarded.

Another way to circumvent the problem is to turn the undefinedness checks off temporarily. For instance, suppose the query `?- ?X:a` causes unintended undefinedness error in module `foo`. Then we can avoid the problem by posing the following query instead:

```
flora2 ?- Method[%_mustDefine(off(foo))]@_sys,
          ?X:a,
          Method[%_mustDefine(on(foo))]@_sys.
```

A more selective way to circumvent this problem is to turn off undefinedness checking just for the offending classes. For instance,

```
?- Method[%_mustDefine(off,?:a@ _@)]@_sys.
```

The fourth way is to deal with the exception is to use $\mathcal{F}$LORA-*2*'s `catch{...}` builtin (note the curly braces):

```
#include "flora_exception.flh"

?- catch{?X:a, FLORA_UNDEFINED_EXCEPTION(?,?)@_prolog, true}.
```

Undefinedness check and update operators. Although undefinedness checking can be turned on and off at will, it cannot always capture all cases correctly. Namely, if an insert or delete statement is executed while undefinedness checking is off, the corresponding methods will not be properly captured an spurious undefinedness errors might result. For instance, if

```
?- insert{a[meth->b]}, delete{a[meth->b]}.
?- Method[%_mustDefine(on)]@_sys.
```

are executed then the query `flora2 ?- a[meth->b]` will cause the undefinedness error. However,

```
?- insert{a[meth->b]}, delete{a[meth->b]}.
?- Method[%_mustDefine(on)]@_sys.
?- a[meth->b].
```

will not flag the method `meth` as undefined.

26.2 Type Checking

Although $\mathcal{F}$LORA-*2* allows specification of object types through signatures, type correctness is not checked automatically. A future versions of $\mathcal{F}$LORA-*2* might support some form of run-time type checking. Nevertheless, run-time type checking is possible even now, although you should not expect any speed here and this should be done during debugging only.

Run-time type checking is possible because F-logic naturally supports powerful meta-programming, although currently the programmer has to do some work to make type checking happen. For instance, a 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:

```
type_error(?O,?M,?R,?D) :-
       %% Values that violate typing
       ?O[?M->?R], ?O[?M=>?D], not ?R:?D
       ;
       %% Defined methods that do not have type information
       ?O[?M->?R], not ?O[?M=>?_D].
?- type_error(?Obj,?Meth,?Result,?Class).
```

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

In this way, one can easily construct special purpose type checkers. This feature is particularly important when dealing with *semistructured* data. (Semistructured 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 portions of the object structure.) In this situation, one might want to limit type checking only to certain methods and classes, because other parts of the data might not be expected to have regular structure.

Note that in a multi-module program, the module information should be added to the various parts of the above type-checker. It is reasonable to assume that the schema information and the definition for the same object resides in the same module (a well-designed program is likely to satisfy this requirement). In this case, a type-checker that take the module information into account can be written as follows:

```
type_error(?O,?M,?R,?D) :-
       %% Values that violate typing
       (?O[?M->?R], ?O[?M=>?D])@?Mod1, not ?R:?D@?Mod1
       ;
       %% Defined methods that do not have type information
       (?O[?M->?R], not ?O[?M=>?_D])@?Mod1.
?- type_error(?Obj,?Meth,?Result,?Class).
```

We should note that type-checking queries in $\mathcal{F}$LORA-*2* are likely to work only for "pure" queries, *i.e.*, ones that do not involve built-ins like arithmetic expressions. Built-ins pose a problem because they typically expect certain variable binding patterns when these built-ins are called. This assumption may not hold when one asks queries as general as `type_error`.

To facilitate all these checks, $\mathcal{F}$LORA-*2* provides a method, `%_check`, in class `Type` of module `_typecheck`. Its syntax is:

```
?- Type[%_check(?Atom,?Result)]@_typecheck.
```

The `?Atom` variable must be bound to an atomic F-logic molecule as described below. `?Result` gets bound to the evidence of type violation (one or two atoms that violate the typing constraint).

- If `?Atom` is of the form `?[?Meth->?]@?Mod` then all type constraints for `?Meth` are checked in module `?Mod`. Missing types (semistructured data) are flagged. If `?Mod` is a variable, then

the constraints are checked in all modules. `?Meth` can also be a variable. In this case all non-procedural methods will be checked.

- If `?Atom` is of the form `?[?Meth=>?]@?Mod` then the type constraints for `?Meth` are checked in module `?Mod` but missing types (semistructured data) are ignored. As before `?Mod` and `?Meth` can be variables.
- If `?Atom` is of the form `?[?Meth*->?]@?Mod` then only the consistency between `*->` and `*=>` is checked. The `->`-style molecules are ignored. Missing types (semistructured data) are flagged.
- If `?Atom` is of the form `?[?Meth*=>?]@?Mod` then again only the consistency between `*->` and `*=>` is checked. The `->`-style molecules are ignored. Missing types (semistructured data) are ignored.

For example, if our knowledge base consists of:

```
a[b->c].
a[b=>d].
c:d.
```

then the query will fail, as the typing is correct:

```
?- Type[%_check(?[?Meth->?],?Result)]@_typecheck.
```

But if, in addition, we had

```
a[b->e].
a[foo->e].
```

then the above query would yield multiple evidences of type inconsistency:

```
?Result = (${a[b -> e]}, ${a[b => d]})
?Result = ${a[foo -> e]}
```

The first means that the atom `a[b -> e]` violates the type constraint specified by the signature `a[b => d]`. The second means that the specified atom does not have a corresponding signature. On the other hand,

```
?- Type[%_check(?[?Meth=>?],?Result)]@_typecheck.
```

will yield only the first evidence because `a[foo->e]` does not violate any typing constraints for semistructured data.

If the object position in the first argument of `%_check` is bound then this object is treated as a class and only the objects in that class will be type-checked. For instance, if we also had

```
q[foo->bar].
q:qq.
```

in our knowledge base then the query

```
?- Type[%_check(qq[?Meth->?],?Result)]@_typecheck.
```

will return one evidence of type inconsistency:

```
?Result = ${q[foo -> bar]}
```

because `q` is the only object in class `qq` that has type violations.

An easy way to remember which type of atoms represent what kind of type checking is to think that `=>` represents typing and, therefore the `=>`-style atoms mean that only the methods that have typing information will be type-checked. The `->`-style atoms, on the other hand, mean that all methods will be checked—whether they have signatures or not. Similarly, `*->` and `*=>` means that only the default values (represented by `*->`) will be checked (and `*=>`, in addition, restricts the checks to the methods that have type information).

26.3 Checking Cardinality of Methods

*F*LORA-*2* does not automatically enforce the cardinality constraint specified in method signatures. However, the `type` system module in *F*LORA-*2* provides methods for checking cardinality constraints for methods that have such constraints declared in their signatures.

In practice as well as in theory things are more complicated however. First, it is theoretically impossible to have a query that will flag a violation of a cardinality constrain if and only if one exists *and* will terminate.

In practice, the constraint checking methods in the `type` system library may trigger run-time errors if there are rules that use non-logical features or certain builtins in their bodies. Therefore, in practice, the user should use the constraint-checking methods only for purely logical methods. Cardinality constraints declared for methods that are defined with the help of non-logical features should be used for documentation only.

The above problems aside, it is easy to verify that a particular satisfies a cardinality constraint. For instance, if method `foo` is declared as

```
someclass[foo {2:3}*=> sometype].
```

then to check that the cardinality constraint is not violated, one can ask the following query:

```
flora2 ?- Cardinality[%_check(?Obj[foo =>?])]@_typecheck.
```

If no violations are found, the above query will *fail*. If there *are* violations of this constraint then `?Obj` will get bound to the objects for which the violation was detected. For instance, consider the following knowledge base:

```
cl[foo {2:3}*=> int].
c::cl.

o1:c.
o2:c.
o3:c.

o1[foo->{1,2,3,4}].
o3[foo->{3,4}].

c[foo *-> 2].
cl[foo *-> {3,4,5}].
```

Then the query

```
?- Cardinality[%_check(?O[foo=>?])]@_typecheck.
```

will return `?O = o1` and `?O = o2` because `o1` has a non-inheritable method `foo` with four values while at most 3 are allowed according to the signature. The object `o2` is returned because `foo` has no values for that object, while at east 2 are required. The object `o3` is not returned because it does not violate the constraint. Similarly, the query

```
?- Cardinality[%_check(?O[foo*=>?])]@_typecheck.
```

will return `?O = c` because the inheritable version of method `foo` has only 1 value for that class, while at least two are required by the signature. The class `cl` is not returned because it does not violate the constraint.

In general, the allowed forms of the method `%_check` in class `Cardinality` are as follows. The argument is always a signature atom (no need to specify reification ${...}). The method type of the signature can be either `=>` or `*=>`. The `=>` version checks non-inheritable methods. Such a method would normally be declared as a class property for a particular class or it would be inherited by an object from its superclass. The `*=>` version checks inheritable methods. Such a method would be either declared as inheritable in a class-object or it would be inherited by a subclass of that class. Whenever an evidence of type violation is required as an answer, the corresponding component of the signature should be specified as an unbound variable. For instance,

- `Cardinality[%_check(?Object[?Method => ?])]@_typecheck`
  Checks cardinality constraints for `?Method` of type `=>` in the current module.
- `Cardinality[%_check(?Obj[?Method *=>?]@?Module)]@_typecheck`
  Checks cardinality constraints for `?Method` of type `*=>` in module `?Module`. If `?Module` is unbound and a cardinality constraint violation is detected in some module then `?Module` is bound to that module.
- `Cardinality[%_check(?Obj[?Method {?LoBound:?HiBound} => ?]@?Mod)]@_typecheck`
  Like the previous query, but the variables `?LoBound` and `?HiBound`, which must be unbound

variables, can be used to indicate which bounds are violated. If the lower bound is violated, then `?LoBound` will be bound to the violated lower bound; otherwise, it is bound to `ok`. If the higher bound is violated, then `?HiBound` is bound to the violated higher bound; otherwise it is bound to `ok`.

If `?Mod` is unbound then it will be bound to the module(s) in which the cardinality constraint is violated.

For instance, for the above program the query

```
?- Cardinality[%_check(?O[foo {?Low:?High} *=> ?]@?Module)]@_typecheck.
```

will bind `?O` to `c`, `?Mod` to `main`, `?Low` to 2, and `?High` to `ok`. Indeed, only the lower bound of the cardinality constraint `c[foo 2:3*=> int]` (which was inherited from `c1`) is violated by the class `c`.

```
?- Cardinality[%_check(?O[foo {?Low:?High} => ?])]@_typecheck.
```

will return the following results:

```
?O = o1
?Low = ok
?High = 3

?O = o2
?Low = 2
?High = ok
```

26.4 Logical Assertions that Depend on Procedural and Non-logical Features

On page 82 we mentioned the potential problems when tabled predicates or F-logic molecules depend on updates. A similar problem arises when such statements depend on non-logical features, such as `var(...)` or on statements that have side effects, such as I/O operations (e.g., `write('foo bar')@_prolog`). Since tabled statements in $\mathcal{F}$LORA-*2* are considered purely logical, one cannot assume that the evaluation happens in the same way as in Prolog. For instance, consider the following program:

```
?O[bar] :- ?O:foo.
?O:foo :- writeln('executed')@_prolog.
?- abc[bar].
```

Despite what one might expect, the above query will cause "executed" to be printed twice — once when `abc[bar]` will be proved for the first time and once when the system will attempt some other way of proving `abc[bar]`. (The system may not realize that the second proof is not necessary.) In general, procedural and side-effectful statements might be executed even if the attempt to prove the statement in the rule head ultimately fails.

$\mathcal{F}$LORA-*2* issues warnings when it finds that a tabled predicate depends on non-logical or side-effectful statements, but it does not warn about all Prolog predicates of this kind. Therefore, caution needs t be exercised in specifying purely logical statements and warnings should not be ignored. If you are certain that a particular suspicious dependency is harmless, use the `ignore_depchk` directive to suppress the warning.

27 Optimizations

27.1 Manual Optimizations

Left-to-right processing. The first rule in improving the performance of $\mathcal{F}$LORA-*2* programs is to remember that query evaluation proceeds from left to right. Therefore it is generally advisable to place subgoals with smaller answer sets as close to the left of the rule body as possible. And, like in databases, Cartesian products should be avoided at all costs.

Nested molecules and path expressions. $\mathcal{F}$LORA-*2* compiler makes decisions about where to place the various parts of complex F-logic molecules, and the programmer can affect this placement by writing molecules in various ways. For instance,

```
?- ..., ?X[attr1 -> ?Y, attr2 -> ?Y], ...
```

is translated as

```
?- ..., ?X[attr1 -> ?Y], ?X[attr2 -> ?Y], ...
```

so the first attribute will be computed first. If the second attribute has a smaller answer set, the attributes in the molecule should be written in the opposite order. The other consideration has to do with literals that have nested molecules in them. For instance, the following query

```
?- ..., ?X[attr1->?Y[attr2->?Z]], f(?P[attr3->?Q]), ...
```

is translated as

```
?- ..., ?X[attr1->?Y], ?Y[attr2->?Z], f(?P), ?P[attr3->?Q], ...
```

i.e., the nested literals follow their hosts in the translation. Thus, writing terms in this way is considered a hint to the compiler, which indicates that bindings are propagated from `?X[attr1->?Y]` to `?Y[attr2->?Z]`, etc. If, on the other hand, `?Y[attr2->?Z]` has only one solution then, perhaps, writing `?Y[attr2->?Z], ?X[attr1->?Y]` might produce a more efficient code. The same considerations apply to `f(?P[attr3->?Q])`.

Similarly to nested molecules, the $\mathcal{F}$LORA-*2* compiler assumes that path expressions represent a hint that bindings are propagated left-to-right. In other words, in `?X.?Y.?Z`, `?X` will be bound

first. Based on this, the oids, of the objects `?X.?Y` are computed, and then the attribute `?Z` is applied. In other words, the translation will be `?X[?Y->?Newvar1], ?Newvar1[?Z->?Newvar2]`.

Unfortunately, unlike in databases, statistical information is not available to the *FLORA-2* compiler and only a few heuristics (such as variable binding analysis, which the compiler does not perform) can be used to optimize such queries. If the order chosen by the compiler is not right, the programmer can always unnest the literals and place them in the right order in the rule body.

Open calls vs. bound calls. In Prolog it is much more efficient (space- and time-wise) to make one unbound call than multiple bound ones. For instance, suppose we have a class, `cl`, that has hundreds of members, and consider the following query:

```
flora2 ?- ?X:cl[attr->?Y].
```

Here, Prolog would first evaluate the open call $?X : cl$ and then for each answer `x` for `?X` it will evaluate `x[attr->?Y]`. If the cost of computing `x[attr->?Y]` is higher than the cost of $x : cl$ and the number of answers to `?X[attr->?Y]` is not significantly higher than the number of answers to `?X:cl`, then the following query might be evaluated much faster:

```
flora2 ?- ?X[attr->?Y], ?X:cl.
```

In this query, a single call `?X[attr->?Y]` is evaluated first and then `x:cl` is computed for each answer for `?X`. Since, as we remarked, the cost of this call can be much smaller than the combined cost of multiple calls to `x[attr->?Y]` for different `x`. If the number of bindings for `?X` in `?X[attr->?Y]` that are not members of class `cl` is small, the second query might take significantly less space and time.

27.2 Invoking the *FLORA-2* Runtime Optimizer

FLORA-2 has a rudimentary runtime optimizer, which can be invoked by executing the following commands:

```
flora2 ?- _optimize(?OptimizerOption)
flora2 ?- _optimize(?OptimizerOption,?Module)
```

The first command invokes the optimizer with the option `?OptimizerOption` in all modules and the second command does the same in a given module. The commands

```
flora2 ?- _resetoptimization(?OptimizerOption)
flora2 ?- _resetoptimization(?OptimizerOption,?Module)
```

disables the corresponding optimizer options.

It should be noted that different queries require different optimizations and any given option might improve the performance of one set of queries while degrading the performance of others.

Some queries may not even work with certain optimizations and produce runtime errors. Therefore, FLORA-2 gives the user the means to turn the optimizations on and off depending on the situation.

At present, FLORA-2 has only two optimization option:

```
local_override
class_expressions
```

Invoking `_optimize(local_override,somemodule)` can, in some cases, speed up query processing by the factor of 10. This optimization typically helps programs that use inheritance, but inheritance is overwritten by locally defined methods in most of the cases.

Invoking `_optimize(class_expressions,somemodule)` always improves performance, sometimes significantly and sometimes negligibly. However, this is done at the expense of disabling the subtype relationships that involve class expressions (see Section 6). So, while `class_expressions` optimization is on the usual subclass relationships among these expressions does not hold. For instance, `c::(c;d)` and `(c,d)::c` are false.

28 Compiler Directives

Executable vs. compile-time directives. Like Prolog compiler, FLORA-2 compiler can take compiler directives. Like in Prolog, these directives can be *executable* or *compile-time*, and this distinction is very important. Executable directives are treated as queries and they begin with `?-`. Compile-time directives begin with `:-`.

Executable directives are mostly used to control how the FLORA-2 shell interprets the expressions that the user types in. These directives have no effect during the compilation of program files. Instead, when they are executed as queries they affect the shell. In contrast, compile-time directives affect the compilation of the files they occur in. Also, if a module is loaded into the main module in the shell, then all compile time directives in that module are executed in the shell as well, so there is no need to explicitly execute these directives in the shell. FLORA-2 requires that all compile-time directives appear at the top of the program prior to the first appearance of a rule or a fact, because such a directive has effect only after it is found and processed.

To better understand the issue, consider the following simple program (say, in file `test.flr`):

```
:- _op(400,xfx,fff).
a fff b.
?- ?X fff ?Y.
```

If you load this program then it will execute correctly and return the bindings `a` and `b` for `?X` and `?Y`, respectively. If you execute the same query `?X fff ?Y` in the FLORA-2 shell, the result will still be correct because FLORA-2 made sure that the directive `_op(400,xfx,fff)` in `test.flr` was executed in the shell as well. On the other hand, if the program was

```
?- _op(400,xfx,fff).
```

```
a fff b.
?- ?X fff ?Y.
```

then `fff` would be known to the shell, but, unfortunately, we will not get that far to find out: The compiler will issue an error, since `fff` will not be known as an operator during the compilation of the program.

Summary of directives. The following is a summary of all supported compiler directives:

`:- setsemantics{`*`Option1, Option2, ...`*`}`
Sets the semantic options in the current user module. The currently allowed options are:

`equality(none)` (default), `equality(basic)`,
`inheritance(none)`, `inheritance(flogic)`,
`custom(none)` (default), and `custom(`*`filename`*`)`.

With `equality(none)`, equality is not maintained, and the symbol `:=:` works like an ordinary predicate. With `equality(basic)`, the predicate `:=:` is treated as equality.

`?- setsemantics{`*`Option1, Option2, ...`*`}`
This is an executable version of the `setsemantics` directive.

`?- setsemantics{`*`Option1, Option2, ...`*`}@`*`module`*
Same as above, except that equality maintenance is set for the specified user module.

`:- index` *`Arity-Argument`*
Says that all tabled HiLog predicates of arity `Arity` should be indexed on argument number `Argument` (the count starts at 1). This directive should appear at the beginning of a module to have any effect. Normally predicates in $\mathcal{F}$LORA-*2* are indexed on predicate name only. The above directive changes this so that indexing is done on the given argument number instead.

Note that the `index` directive is not very useful for predicates that mostly contain facts, because these are trie-indexed anyway (regardless of what you say). Thus, this instruction is useful only for predicates with partially instantiated arguments that appear in the rule heads.
This is an executable version of the `index` directive. The module of the predicates can be specified.

`:- index` *`%Arity-Argument`*
The `index` directive for non-tabled HiLog predicates.

`?- (index` *`%Arity-Argument`*`)[@`*`module`*`]`
The executable `index` for non-tabled HiLog predicates.

`:- _op(`*precedence*`,`*type*`,`*operator*`)`
Defines *operator* as a $\mathcal{F}$LORA-*2* operator with the given precedence and type. The *type* is the same as in Prolog operators, *i.e.*, `fx`, `xf`, `xfy`, etc. Note that the `op` directive is confined to the module in which it is executed or defined. For instance, if `example.flr` has a call `(a`

`foo b)@bar`, the symbol `foo` is declared as an operator in the program loaded to module `bar`, but not in `example.flr`, then a syntax error will result, because `example.flr` does not know about the operator declaration for `foo`.

`:-` `_op(`*precedence*`,`*type*`,[`*operator*`, ..., `*operator*`])`
Same as above, except that this directive defines a list of operators with the same precedence and type.

`?-` `_op(`*precedence*`,`*type*`,`*operator*`)@`*module*
Same as above, except that a module is also given. However, unless the module is `main`, this directive acts as a no-op.

`table` *functor/arity*`, ...,` *functor/arity*
Requests that the specified first order predicates must be tabled.

29 FLORA-2 System Modules

FLORA-2 provides a number of useful libraries that other programs can use. These libraries are statically preloaded into modules that are accessible through the special `@_modname` syntax, and they are called *system modules*. We describe the functionality of these modules below. Some of these modules also have longer synonyms. These synonyms are mentioned below, if they exist.

29.1 Input and Output

This library simplifies access to the most common Prolog I/O facilities. This library is preloaded into the system module `io` and can be accessed using the `@_io` syntax.

The purpose of the I/O library is not to replace the standard I/O predicates with FLORA-2 methods, but rather to relieve the user from the need to do explicit conversion of arguments between the HiLog representation of terms used in FLORA-2 and the standard Prolog representation of the underlying engine.[12] However, for uniformity, the `io` library also provides certain methods that do not suffer from the conversion problem.

The library contains two types of I/O operations: *stream-based I/O* and *port-based*. Stream-based I/O is based on the standard Prolog I/O primitives. It uses symbols as file handles. Port-based I/O is specific to XSB. Its file handles are internally represented as numbers. Although stream-based I/O is often easier to use, there are many more port-based primitives that can accomplish various low-level I/O operations. This FLORA-2 library provides just a few common ones. See the XSB manual, volume 2, for a complete list of these primitives.

The methods and predicates accessible through the `io` library are listed below. Note that some operations are defined as procedural methods and others as predicates. This is because we use the object-oriented representation only where it makes sense — we avoid introducing additional classes and objects that require more typing just for the sake of keeping the syntax object-oriented.

[12] See Section 12 for a discussion of the problems associated with this representation mismatch.

Stream-based I/O.

- `%see(?Filename), ?Filename[%see]` — open `?Filename` and make it into the current input stream.
- `%seeing(?Stream)` — binds `?Stream` to the current input stream.
- `%seen` — closes the current input stream.
- `%tell(?Filename), ?Filename[%tell]` — opens `?Filename` as the current output stream.
- `%telling(?Stream)` — binds `?Stream` to the current output stream.
- `%told` — closes the current output stream.
- `%write(?Obj)` — writes `?Obj` to the current output stream.
- `?Stream[%write(?Obj)]` — writes `?Obj` to the stream `?Stream`.
- `%writeln(?Obj)`, `?Stream[%writeln(?Obj)]` — same as above, except that the newline character is output after `?Obj`.
- `%nl` — writes the newline character to the current output stream.
- `%read(?Result)` — binds `?Result` to the next term in the current input stream.
- `?Stream[%read(?Result)]` — same as above, but use `?Stream` as the input stream.

Port-based I/O.

- `?Filename[%open(?Mode,?Port)]` — opens `?Filename` with mode `?Mode` (which can be `r`, `w`, or `a`) and binds `?Port` to the file handle.
- `?Port[%close]` — closes the file handle to which `?Port` is bound.
- `?Port[%read(?Result)]` — bind `?Result` to the next term in the previously open input `?Port`.
- `%stdread(?Result)` — same, but use the standard input as the port.
- `?Port[%write(?Result)]` — write `?Result` out to the previously open output `?Port`.
- `%stdwrite(?Result)` — same, but use standard output as the port.
- `%fmt_write(?Format,?Term)` — C-style formatted output to the standard output. See the XSB manual, volume 2, for the description of all the options.
- `?Port[%fmt_write(?Format,?O)]` — same, but use `?Port` for the output.
- `%fmt_write_string(?String,?Format,?Obj` — same as above, but bind `?String` to the result. See the XSB manual for the details.

- `%fmt_read(?Format,?Result,?Status)` — C-style formatted read from standard input. See the XSB manual.
- `?Port[%fmt_read(?Format,?Result,?Status)]` — same, but use `?Port` for input.
- `%write_canonical(?Term)` — write `?Term` to standard output in canonic Prolog form.
- `?Port[%write_canonical(?Term)]` — same, but use `?Port` for output.
- `%read_canonical(?Term)` — read standard input and bind `?Term` to the next term in the input. The term *must* be in canonical Prolog form, or else an error will result. This method is much faster than the usual `read` operation, but it is not as versatile, as it assumes that input is in canonical form.
- `?Port[%read_canonical(?Term)]` — same, but use `?Port` for input.
- `%readline(?Type,?String)` — read the standard input and bind `?String` to the next line. `?Type` is either `atom` or `charlist`. The former means that `?String` is to be bound to a Prolog atom and the latter binds it to a list of characters.
- `?Port[%readline(?Type,?String)]` — same, but use `?Port` for input.

Common file operations. The `_io` module also provides a class `File`, which has methods for the most common file operations. These include:

- File[%exists(?F)]. True if file `?F` exists.
- File[%readable(?F)]. True if file `?F` is readable.
- File[%writable(?F)]. True if the file is writable.
- File[%executable(?F)]. True if the file is executable.
- File[%modtime(?F,?T)]. Binds `?T` to the last modification time of `?F`.
- File[%mkdir(?F)]. Makes a directory named after the value of `?F`.
- File[%rmdir(?F)]. Removes the directory `?F`.
- File[%chdir(?F)]. Changes the current directory to `?F`.
- File[%cwd(?F)]. Binds `?F` to the current working directory in the shell.
- File[%link(?F,?Dest)]. Creates a link named after `?F` to the existing file `?Dest`.
- File[%unlink(?F)]. Removes the link `?F`.
- File[%remove(?F)]. Removes the file `?F`.
- File[%tmpfilename(?F)]. Binds `?F` to a temporary file with a completely new name.
- File[%isabsolute(?F)]. True if `?F` is an absolute path name.

- File[%rename(?F,?To)]. Renames file `?F` to file `?To`.
- File[%basename(?F,?Base)]. Binds `?Base` to the base name of file path `?F`. For instance, `?- File[%basename('/a/b/cde',?Base)]` would bind `?Base` to `cde`.
- File[%extension(?F,?Ext)]. Binds `?Ext` to the extension of the file ?F. For instance, `?- File[%extension('/a/b/cde.exe',?Ext)]` would bind `?Ext` to `exe`.
- File[%dirname(?F,?Dir)]. Binds `?Dir` to the directory name of file `?F`.
- File[%expand(?F,?Expanded)]. Expands the file `?F` by attaching the directory name (if the file is not absolute) and binds `?Expanded` to that expansion.
- File[%newerthan(?F,?F2)]. True if `?F` is a newer file than `?F2`.
- File[%copy(?F,?To)]. Copies the contents of the file ?F to `?To`.

29.2 Storage Control

FLORA-2 keeps the facts that are part of the program or those that are inserted by the program in special data structures called *storage tries*. The system module `db` accessible through the module reference `@_db`, provides primitives for controlling this storage. This module also has a longer synonym `_storage`.

- `%commit` — commits all changes made by transactional updates. If this statement is executed in the middle of an update transaction, changes made by transactional updates prior to this will be committed and will not be undone even if a subsequent subgoal fails.
- `%commit(?Module)` — commits all changes made by transactional updates to facts in the user module `?Module`. Backtrackable updates to other modules are unaffected.
- `%purgedb(?Module)` — deletes all facts previously inserted into the storage associated with module `?Module`.

29.3 System Control

The system module `sys` provides primitives that affect the global behavior of the system. It is accessible though the system module reference `@_sys` (or through its synonym `_system`).

- `Libpath[%add(?Path)]` — adds `?Path` to the library search path. This works similarly to the `?PATH` environment variable in that when the compiler or the loader are trying to locate a file specified by its name only (without directory, etc.) then they examine the files stored in the directories on the library search path.
- `Libpath[%remove(?Path)]` — removes `?Path` from the library search path.

- `Libpath[%query(?Path)]` — queries the library search path. If `?Path` is bound, checks if the specified directory is on the library search path. Otherwise, binds (through backtracking) `?Path` to each directory on the library search path.
- `Tables[%abolish]` — discards all tabled data in Prolog.

This module also provides the following amenities:

- `%abort(?Message)` — puts `?Message` on standard error stream and terminates the current execution. Message can also be in the form `(?M1, ?M2, ..., ?Mn)`. In this case, all the component strings are concatenated before printing them out.

  User aborts can be caught as follows:

  ```
  ?- catch{?Goal, FLORA_ABORT(FLORA_USER_ABORT(?Message),?_), ?Handler}
  ```

  In order to be able to use the predefined constants `FLORA_ABORT` and `FLORA_USER_ABORT` the program must contain the include statement

  ```
  #include "flora_exceptions"
  ```

- `%warning(?Message)` — prints a warning header, then message, `?Message`, and continues. Output goes to standard error stream. `?Message` can be of the form `(?M1, ?M2, ..., ?Mn)`.
- `%message(?Message)` — Like `warning/1`, but does not print the warning header. `?Message` can be of the form `(?M1, ?M2, ..., ?Mn)`.

29.4 Cardinality Constraint Checking

This system module of FLORA-2 provides methods for testing cardinality constraints of the methods defined in the FLORA-2 knowledge base. The module defines the transactional method `%_check` in class `Cardinality` of module `_typecheck`. This method is described in Section 26.3.

29.5 Data Types

This system module of FLORA-2 provides methods for accessing the components of data types such as `_dateTime`, `_iri`, and so on. Data types are described in Section 25.

29.6 Reading and Compiling Input Terms

Sometimes it may be necessary for an application to read and compile FLORA-2 statements from an input source. To this end, the `_parse` system library provides the following predicates and methods.

- `?- %read(?Code,?Mod,?Stat)@_parse.`

Read the next term from the standard input and compile it. The resulting term is bound to the variable `?Code`. The term can also be a reified formula and even a reified rule. Such a formula/rule can be used in a query or inserted into the knowledge base as appropriate. If the input term is not reified, the `?Mod` parameter has no effect. If the formula is reified, them it will be built for the module specified in `?Mod`. If `?Mod` is unbound then the default module `main` is assumed.

`?Status` is bound to the status code returned by the predicate and has the form `[OutcomeFlag, EOF_flag|ErrorList]`, where:

OutcomeFlag = null/or/error
null - a blank line was read, no code generated (?Code = null)
ok - good code was generated, no errors
error - parsing/compilation errors
EOF_flag = eof/not_eof
not_eof - end-of-file has not been reached
eof - if it has been reached.
ErrorList - if OutcomeFlag=null/ok, then this list would be empty.
if OutcomeFlag=error, then this would be a list of the form `[error(N1,N2,Message), ...]`, where N1, N2 encode the line and character number, which is largely irrelevant in this context. Message is an error message. Error messages are displayed.

Example:

```
?- %read(?Code,foobar,?Stat)@_parse.
f(a).   <-----user input

?Code = f(a)
?Stat = [ok, not_eof]

?- %read(?Code,foobar,?Stat)@_parse.
${a[b->c]}.   <-----user input

?Code = ${a[b -> c]@foobar}
?Stat = [ok, not_eof]
```

- `?- %readAll(?Code,?Mod,?Stat)@_parse.`

Used for reading terms one-by-one and returning answers interactively. The meaning of the arguments is the same. Under one-at-a-time solution (`_one`), will wait for input, return compiled code, then wait for input again, if the user types ";". If the user types `RET` then this predicate succeeds and exits. Under all-solutions semantics (`_all`), will wait for inputs and process them, but will not return answers unless the file is closed (e.g., Ctl-D at standard input).

- `?- ?Source[%readAll(?Module,?CodeList)]@_parse.`

This collects all answers from a source, which can be either a file or a string. If the source is a string, it should be specified as `string(Str)`, where `Str` is either an atom or a variable that is bound to an atom. If the source is a file then it should be specified as `file(FileName)`, where `FileName` is an atom that specifies a file name (or a variable bound to it). The meaning of `?Module` is the same as before. `?CodeList` contains the result. It is bound to a list of the form `[code(TermCode1,Status1), code(TermCode2,Status2), ...]`, where `TermCode` is the compiled code of a term in the source, and `?Status` is the status of the compilation for this term. It has the form `[OutcomeFlag, EOF_flag|ErrorList]`, as explained before.

30 Notes on the Programming Style and Common Pitfalls

Programming in $\mathcal{F}$LORA-*2* is similar to programming in Prolog, but is more declarative. For one thing, F-molecules are always tabled, so the programmer does not need to worry about tabling the right predicates. Second, there is no need to worry that a predicate must be declared as dynamic in order to be updatable. Third — and most important — the facts specified in the program are considered to be part of the database. In particular, their order does not matter and duplicates are eliminated automatically.

30.1 Facts are Unordered

The fact that $\mathcal{F}$LORA-*2* does not assume any particular order for facts has a far-reaching implication on the programming style and is one of the pitfalls that a programmer should avoid. In Prolog, it is a common practice to put the catch-all facts at the end of a program block in order to capture subgoals that do not match the rest of the program clauses. For instance,

```
p(f(?X)) :- ...
p(g(?X)) :- ...
%% If all else fails, simply succeed.
p(?_).
```

This will *not* work in $\mathcal{F}$LORA-*2*, because `p(?_)` will be treated as a database fact, which is placed in no particular order with respect to the program. If you want the desired effect, represent the catch-all facts as rules:

```
p(f(?X)) :- ...
p(g(?X)) :- ...
%% If all else fails, simply succeed. Use rule notation for p/1.
p(?_) :- true.
```

30.2 Testing for Class Membership

In imperative programming, users specify objects' properties together with the statements about the class membership of those objects. The same is true in $\mathcal{F}$LORA-*2*. For instance, we would

specify an object `John` as follows, which is conceptually similar to, say, Java:

```
John : person
[ name->'John Doe',
  address->'123 Main St.',
  hobby->{chess, hiking}
].
```

However, in *FLORA-2* attributes can also be specified using rules. For instance, we can say that (in our particular enterprise) an employee works in the same building where the employee's department is located:

```
?X[building->?B] :- ?X:employee[department->?_[building->?B]].
```

Our experience in teaching F-logic programming to users indicates that initially there is a tendency to confuse premises with consequents when it comes to class membership. So, a common mistake is to write the above as

```
?X:employee[building->?B] :- ?X[department->?[building->?B]].
```

A minute reflection should convince the reader that his is incorrect, since the above rule is equivalent to two statements:

```
?X[building->?B] :- ?X[department->?_[building->?B]].
?X:employee :- ?X[department->?_[building->?B]].
```

It is the second statement, which is problematic. Certainly, we did not intend to say that any object with a `department` attribute pointing to an object with a `building` attribute is an employee!

It is interesting to note that such a confusion between premises and consequences is common only when it comes to class membership. Therefore, the user should be carefully check the validity of placing class membership molecules in the rule heads.

30.3 Complex F-molecules in the Rule Heads

Another common mistake is the inappropriate use of complex F-molecules in the rule heads. When using a complex molecule, such as `a[b->c, d->e]`, one must always keep in mind that its meaning is `a[b->c] and a[d->e]` whether the molecule occurs in the rule head or in its body. Therefore, if `a[b->c, d->e]` occurs in the head of a rule like

```
a[b->c, d->e] :- body.
```

then the rule can be broken up in two using the usual logical tautology $((\mathtt{X} \wedge \mathtt{Y}) \leftarrow \mathtt{Z}) \equiv (\mathtt{X} \leftarrow \mathtt{Z}) \wedge (\mathtt{Y} \leftarrow \mathtt{Z})$:

```
a[b->c] :- body.
a[d->e] :- body.
```

Forgetting this tautology sometimes causes logical mistakes. For instance, suppose `flight` is a binary relation that represents direct flights between cities. Then a rule like this

```
flightobj[from->?F, to->?T] :- flight(?F,?T).
```

is likely to be a mistake if the user simply wanted to convert the relational representation into an object-oriented one. Indeed, in the head, `flight` is a *single* object and therefore both `from` and `to` will get multiple values and it will not be possible to find out (by querying that object) which cities have direct flights between them. The easiest way to see this is through the use of the aforesaid tautology:

```
flightobj[from->?F] :- flight(?F,?T).
flightobj[to->?T] :- flight(?F,?T).
```

Therefore, if the `flight` relation has the following facts

```
flight(newyork,boston).
flight(seattle,toronto).
```

then the following molecules will be derived (where the last two are unintended):

```
flightobj[from->newyork, to->boston].
flightobj[from->seattle, to->toronto].
flightobj[from->newyork, to->toronto].
flightobj[from->seattle, to->boston].
```

To rectify this problem one must realize that each tuple in the `flight` relation must correspond to a separate object in the rule head. The error in the above program is in that *all* tuples in `flight` correspond to the same object `flightobj`. There are two general ways to achieve our goal. Both try to make sure that a new object is used in the head for each `flight`-tuple.

The first method is to use a new function symbol, say `f`, to construct the oids in the rule head:

```
f(?F,?T):flight[from->?F, to->?T] :- flight(?F,?T).
```

As an added bonus, we also created a class, `flight`, and made the flight objects into the members of that class. While it solves the problem, this approach might not always be acceptable, since the oid essentially explicitly encodes all the information in the tuple.

An alternative approach is to use the `newoid{...}` primitive from Section 10. Here we are using the fact that each time `flight(?F,?T)` is satisfied, `newoid{?X}` generates a new value for `?X`.

```
?O:flight[from->?F, to->?T] :- newoid{?O}, flight(?F,?T).
```

This approach is not as declarative as the first one, but it saves the user from the need to figure out how exactly the oids in the rule head should be constructed.

31 Miscellaneous Features

31.1 Suppression of Banners

When *FLORA-2* initializes itself, it produces a lot of chatter: the XSB banner, a great number of messages about the various XSB and *FLORA-2* modules that are loaded, and the *FLORA-2* welcome message. However, the user might not want to see all these, and when the program runs in a batch mode or interacts with other programs then all these messages are either not needed or they can complicate this interaction. To suppress these messages, use the following switches when invoking the system:

`--nobanner`: suppress the XSB banner and the *FLORA-2* welcome message
`--quietload`: when loading a module, do not print feedback to the terminal

Thus,

```
runflora --nobanner --quietload
```

will get you directly to the *FLORA-2* prompt without cluttering the terminal with chatter.

Sometimes even the prompt stands in the way. For instance, when *FLORA-2* interacts with other programs (*e.g.*, with a GUI) then sending the prompt to the other program just complicates things, as the receiving program needs to remember to ignore the prompt. To avoid this complication, the invocation flag `--noprompt` is provided. Thus,

```
runflora --nobanner --quietload --noprompt
```

will print nothing on startup and will be just waiting for user input. When the input occurs, *FLORA-2* will evaluate the query and return the result. After this, it will return to wait for the input without issuing any prompts.

31.2 Passing Compiler Options to XSB

Sometimes it is desirable to pass compiler options to the underlying Prolog compiler. To do this, *FLORA-2* provides the directive `flora_compiler_options/1`. It takes one argument — a list of options that is understood by the underlying Prolog compiler. For instance, the directive

```
:- flora_compiler_options([spec_repr]).
```

will cause the module that contains this directive to be compiled with the XSB specialization optimization.

32 Bugs in Prolog and *FLORA-2*: How to Report

The *FLORA-2* system includes a compiler and runtime libraries, but for execution it relies on Prolog. Thus, some bugs that you might encounter are the fault of *FLORA-2*, while others are

Prolog bugs. For instance, a memory violation that occurs during the execution is in all likelihood an internal Prolog bug. (FLORA-2 is a stress test — all bugs come to the surface.)

An incorrect result during the execution can be equally blamed on Prolog or on FLORA-2— it requires a close look at the program. A compiler or a runtime error for a perfectly valid program is probably a bug in the FLORA-2 system.

Bugs that are the fault of the underlying Prolog engine are particularly hard to fix, because FLORA-2 programs are translated into mangled, unreadable to humans Prolog code. To make things worse, this code might contain calls to FLORA-2 system libraries.

To simplify bug reporting, FLORA-2 provides a utility that makes the compiled Prolog program more readable. The `_dump/1` predicate can be used to strip the macros from the code, making it much easier to understand. If you issue the following command

```
flora2 ?- _dump(foo).
```

the program `foo.flr` will be compiled without the macros and dump the result in the file `foo_dump.P`. This file is pretty-printed to make it easier to read. Similarly,

```
flora2 ?- _dump(foo,bar)
```

will compile `foo.flr` for module `bar` and will dump the result to the file `foo_dump.P`.

Unfortunately, this more readable version of the translated FLORA-2 program might still not be executable on its own because it might contain calls to FLORA-2 libraries or other modules. The set of guidelines, below, can help cope with these problems.

Reporting FLORA-2-related Prolog bugs. If you find a Prolog bug triggered by a FLORA-2 program, here is a set of guidelines that can simplify the job of the XSB developers and increase the chances that the bug will be fixed promptly:

1. Reduce the size of your FLORA-2 program as much as possible, while still being able to reproduce the bug.
2. Eliminate all calls to the system modules that use the `@_lib` syntax. (Prolog modules that are accessible through the `@_prolog(modname)` syntax are OK, but the more you can eliminate the better.)
3. If the program has several user modules, try to put them into one file and use just one module.
4. Use `_dump/1` to strip FLORA-2 macros from the output of the FLORA-2 compiler.
5. See if the resulting program runs under plain XSB system (without the FLORA-2 shell). If it does not, it means that the program contains calls to FLORA-2 runtime libraries. Try to eliminate such calls.

   One common library call is used to collect all query answers in a list and then print them out. You can get rid of this library call by finding the predicate `fllibprogramans/2` in

the compiled `.P` program and removing it while preserving the subgoal (the first argument) and renaming the variables (as indicated by the second argument). Make sure the resulting program is still syntactically correct!

Other calls that are often no longer needed in the dumped code are those that load FLORA-2 runtime libraries (which we are trying to eliminate!). These calls have the form

```
?- flora_load_library(...).
```

If there are other calls to FLORA-2 runtime libraries, try to delete them, but make sure that the bug is still reproducible.

6. If the program still does not run because of the hard-to-get-rid-of calls to FLORA-2 runtime libraries, then see if it runs after you execute the command

   ```
   ?- bootstrap_flora.
   ```

   in the Prolog shell. If the program runs after this (and reproduces the bug) — it is better than nothing. If it does not, then something went wrong during the above process: start anew.

7. Try to reduce the size of the resulting program as much as possible.

8. Tell the XSB developers how to reproduce the bug. Make sure you include all the steps (including such gory details as whether it is necessary to call `bootstrap_flora/0`).

Finally, remember to include the details of your OS and other relevant information. Some bugs might be architecture-dependent.

Reporting FLORA-2 bugs. If you believe that the bug is in the FLORA-2 system rather than in the underlying Prolog engine, the algorithm is much simpler:

1. Reduce the size of the program as much as possible by deleting unrelated program clauses and squeezing a multi-module program into just one file.

2. Remove all the calls to system modules, unless such a call that is the essence of the bug.

3. Tell FLORA-2 developers how to reproduce the bug.

The current version contains the following known bugs, which are due to the fact that certain features are yet to be implemented:

1. Certain programs might cause the following XSB error message:

   ++Error[XSB]: [Compiler] '!' after table dependent symbol

This is due to certain limitations in the implementation of tabled predicates in the XSB system. This problem will be eliminated in a future release of XSB. Meanwhile, as explained in the Introduction, configuring XSB for SLG-WAM and *local* scheduling will avoid many of such errors.

2. Error messages when FLORA-2 update predicates contain arithmetic expressions in the query part. This problem will be fixed in the future.

3. Inheritance of procedural methods is not supported: `a[*%p(?X)]`.

Appendices

A A BNF-style Grammar for FLORA-2

```
%% To avoid confusion between some language elements and meta-syntax
%% (e.g., parentheses and brackets are part of BNF and also of the language
%% being described), we enclose some symbols in single quotes to make it
%% clear that they are part of the language syntax, not of the grammar.
%% However, in FLORA these symbols can be used with or without the quotes.

Rule := Head (':-' Body)? .

Query := '?-' Body.

Directive := ':-' ExportDirective | OperatorDirective | SetSemanticsDirective
             | IgnoreDependencyCheckDirective | PrologDirective

Head := HeadLiteral
Head := Head (',' | 'and') Head

HeadLiteral := BinaryRelationship | ObjectSpecification | Term

Body := BodyLiteral
Body := BodyConjunct | BodyDisjunct | BodyNegative | ControlFlowStatement
Body := Body '@' ModuleName
Body := BodyConstraint

ModuleName := atom | 'atom()' | atom '(' atom ')' | thisModuleName

BodyConjuct  := Body (',' | 'and') Body
BodyDisjunct := Body (';' | 'or') Body
BodyNegative := (('not' | '
+') Body)  |  'false' Body ''
BodyConstraint := '' CLPR-style constraint ''

ControlFlowStatement  := IfThenElse | UnlessDo
                           | WhileDo | WhileLoop
  | DoUntil | LoopUntil
IfThenElse := 'if' Body 'then' Body  ('else' Body)?  | Body '<-' Body
UnlessDo   := 'unless' Body 'do' Body
WhileDo    := 'while' Body 'do' Body
WhileLoop  := 'while' Body 'loop' Body
DoUntil    := 'do' Body 'until' Body
LoopUntil  := 'loop' Body 'until' Body
```

```
BodyLiteral := BinaryRelationship | ObjectSpecification | Term
                | DBUpdate | Refresh | NewoidOp | Builtin | Loading
                | CatchExpr | ThrowExpr | TruthTest

Builtin := ArithmeticComparison, Unification, MetaUnification, etc.

Loading := '[' LoadingCommand (',' LoadingCommand)* ']'
LoadingCommand := filename ('>>' atom)

BinaryRelationship := PathExpression ':' PathExpression
BinaryRelationship := PathExpression '::' PathExpression

ObjectSpecification := PathExpression '[' SpecBody ']'

SpecBody := 'not' MethodSpecification
SpecBody := SpecBody ',' SpecBody
SpecBody := SpecBody ';' SpecBody

MethodSpecification :=  ('%' | '*')? Term
MethodSpecification :=  PathExpression ValueReferenceConnective PathExpression

ValueReferenceConnective :=  '->' | '*->' | '=>' | '*=>'
                                    '+>>' | '*+>>' | '->->' | '*->->'

PathExpression := atom | number | string | variable | specialOidToken
PathExpression := Term | List | ReifiedFormula
PathExpression := PathExpression  PathExpressionConnective PathExpression
PathExpression := BinaryRelationship
PathExpression := ObjectSpecification
PathExpression := Aggregate

PathExpressionConnective :=  '.' | '!'

specialOidToken :=  anonymousOid | numberedOid | thisModuleName

ReifiedFormula := $ (Body | '(' Rule ')')+

%% No quotes are allowed in the following special tokens!
%% No space allowed between _# and integer
%% anonymousOid & numberedOid can occur only in rule head
%% or in reified formulas
anonymousOid   := '_#'
%% No space between _# and integer
numberedOid    := '_#'integer
```

```
thisModuleName := '_@'

List := '[' PathExpression (',' PathExpression)* ('|' PathExpression)? ']'

Term := Functor '(' Arguments ')'

Term := '%' Functor '(' Arguments ')'

Functor := PathExpression

Arguments := PathExpression (',' PathExpression)*

Aggregate := AggregateOperator '' TargetVariable (GroupingVariables)? '|' Body ''
AggregateOperator := 'max' | 'min' | 'avg' | 'sum' | 'collectset' | 'collectbag'
%% Note: only one TargetVariable is permitted.
%% It must be a variable, not a term. If you need to aggregate over terms,
%% as for example, in collectset/collectbag, use the following idiom:
%%      S = collectset V | ... , V=Term
TargetVariable := variable
GroupingVariables := '[' variable, (',' variable)* ']'

DBUpdate := DBOp '' UpdateList ('|' Body)? ''
DBOp := 'insert' | 'insertall' | 'delete' | 'deleteall' | 'erase' | 'eraseall'
UpdateList := HeadLiteral ('@' atom)?
UpdateList := UpdateList (',' | 'and') UpdateList
Refresh := 'refresh' UpdateList ''

RuleUpdate := RuleOp '' RuleList ''
RuleOp := 'insertrule' | 'insertrule_a' | 'insertrule_z' |
               'deleterule' | 'deleterule_a' | 'deleterule_z'
RuleList := Rule | '(' Rule ')'  ( (',' | 'and')  '(' Rule ')' )*

NewoidOp := 'newoid' Variable ''

CatchExpr := 'catch' Body, Term, Body ''
ThrowExpr := 'throw' Term ''
TruthTest := 'true' Body '' | 'unknown' Body '' | 'false' Body ''
```

B The FLORA-2 Debugger

FLORA-2 debugger is implemented as a presentation layer on top of the Prolog 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-2 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.

Tracing. To start tracing, you must issue the command `_trace` at the FLORA-2 prompt. It is also possible to put the subgoal `_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 `_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 controlled by the predicate `debug_ctl`. For instance, executing `debug_ctl(profile, on)` at the FLORA-2 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.

FLORA-2 provides a convenient shortcut that capture some of the most common uses of the aforesaid `debug_ctl` interface. Executing

```
flora2 ?- _trace(filename).
```

will switch FLORA-2 to non-interactive trace mode and the entire trace will be dumped to file `filename`. Note that you have to execute `_notrace` or exit Prolog in order for the entire file to be flushed on disk.

Low-level tracing. FLORA-2 debugger also supports low-level tracing via the shell command `_tracelow`. With normal tracing, the debugger converts low-level subgoals to subgoals that are found in the user program and are thus meaningful to the programmer. With low-level tracing, the debugger displays the actual Prolog subgoals (of the compiled .P program) that are being executed. This facility is useful for debugging FLORA-2 runtime libraries.

As with _trace, FLORA-2 provides a convenient shortcut that allows the entire execution trace to be dumped into a file:

```
flora2 ?- _tracelow(filename).
```

C Emacs Support

Editing and debugging $\mathcal{F}$LORA-*2* programs can be greatly simplified with the help of *flora-mode*, a special Emacs editing mode designed specifically for $\mathcal{F}$LORA-*2* programs. Flora-mode provides support for syntactic highlighting, automatic indentation, and the ability to run $\mathcal{F}$LORA-*2* programs right out of the Emacs buffer.

C.1 Installation

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

```
.../flora2/emacs/flora.el
```

found in your $\mathcal{F}$LORA-*2* 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
```

This will produce the file `flora.elc` — a compiled byte code. If you are using XEmacs, use `xemacs` instead of `emacs` above — the two emacsen use incompatible byte code, and you cannot use `flora.elc` compiled under one system for editing files under another.

Finally, you must tell X/Emacs how to recognize $\mathcal{F}$LORA-*2* 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-*2* programs), you can do the following:

- In Emacs: select `Help.Options.Global Font Lock` on the menubar. To enable highlighting 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).

C.2 Functionality

Menubar menu. Once FLORA-2 editing mode is installed, it provides a number of functions. First, whenever you edit a FLORA-2 program, you will see the "Flora" menu in the menubar. This menu provides commands for controlling the Flora process (i.e., the FLORA-2 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 to the FLORA-2 shell.

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:

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

When you invoke any of the above commands, a FLORA-2 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 FLORA-2 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 FLORA-2 is equivalent to typing `Ctl-c` at the FLORA-2 prompt. Quitting the process stops the Prolog engine, and restarting the process shuts down the old Prolog process and starts a new one with FLORA-2 shell running.

Indentation. Flora editing mode understands some aspects of the FLORA-2 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 $\mathcal{F}$LORA-*2* 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.

D Inside FLORA-2

D.1 How FLORA-2 Works

As an F-logic-to-Prolog compiler, FLORA-2 first parses its source file, compiles it into Prolog syntax and then outputs the resulting code. For instance the command

```
flora2 ?- _notrace(myprog).
```

compiles the program found in the file `myprog.flr` and generates the following files: `myprog.P`, `myprog_main.xwam`, and `myprog.fdb` (if `myprog.flr` contains facts in addition to rules). By default, `_load(myprog)` loads the program into the default user module named `main`. If `myprog.flr` contains F-logic facts, all these facts will be compiled separately into the file `myprog.fdb` that is dynamically loaded at runtime. Next, the file `myprog.P` is generated — take a look at "`myprog_main.P`" to see what has become of your FLORA-2 program! — and passed to the Prolog compiler, yielding Prolog byte code `myprog.xwam`, which is then renamed to `myprog_main.xwam`. This file is then loaded and executed. If `myprog.flr` contains queries, they are immediately executed by Prolog (provided there are no errors).

In the module system of FLORA-2, the same program can be loaded into any user module. The same program can even be loaded into two different modules at the same time, in which case there will be two distinct copies of the same program running at the same time. For each user module, a different byte code is generated (this is why `myprog.xwam` was renamed into an object file that contains the module name as part of the file name).

The main purpose of the FLORA-2 shell is to allow the evaluation of ad-hoc F-logic queries. For example, after consulting and loading the the file `default.flr` from the demo directory by launching the command `?- _demo(default).`, pose the following query and see what happens.

```
flora2 ?-  ?X..kids[                        // Whose kids
                  self -> ?K,               // ... (list them by name)
                  hobbies ->                // ... have hobbies
                    ?H:dangerous_hobby      // ... that are dangerous?
           ].
```

FLORA-2 compilation. The basic idea behind the implementation of F-logic by translating it into predicate calculus is described in [8]. It consists of two parts: translation of F-molecules into various kinds of Prolog predicates, and addition of appropriate "closure rules" that implement the object-oriented semantics of the logic.

Consider, for instance, the following complex F-molecule, which represents some facts about the object `Mary`:

```
Mary:employee[age->29, kids->{Tim,Leo}, salary(1998)->100000].
```

As described in [8], any complex F-molecule can be decomposed into a conjunction of simpler F-logic atomic formulas. These latter atoms can be directly represented using Prolog syntax. For

different kinds of F-logic atoms we use different Prolog predicates. For instance, the result of translating the above F-molecule might be:

```
isa(Mary,employee).              // Mary:employee.
fd(Mary,age,29).                 // Mary[age->29].
mvd(Mary,kids,Tim).              // Mary[kids->{Tim}].
mvd(Mary,kids,Leo).              // Mary[kids->{Leo}].
fd(Mary,salary(1998),100000).    // Mary[salary(1998)->100000].
```

The `mvd` predicate is used to encode methods that return values (as opposed to Boolean methods). The predicates `isa` ans `sub` encode the IS-A and subclass relationships, respectively. We call these predicates *wrapper predicates*. Of course, FLORA-2 has much more: signatures, inheritable and non-inheritable methods, directives, and all kinds of auxiliary predicates needed to improve efficiency.

FLORA-2 facts, such as above, are then stored in a special data structure, called trie, and are retrieved using "patch rules", which have the form

```
fd(Obj,Meth,Val) :- storage_find_fact(TrieName, fd(Obj,Meth,Val)).
```

where `storage_find_fact/2`is an XSB predicate that retrieves facts from tries. `TrieName` is the *storage trie* that is specifically dedicated to storing facts. There is one such trie per module. Since programs are loaded into modules dynamically, the name of the storage trie is determined at program load time. Also, as we shall discuss later, `fd`, `mvd`, etc., are not the actual names of the predicates used in the encoding. The actual names have the module name prepended to them and thus are different for different modules. Moreover, since module names are determined at program load time, the names of the wrapper predicates are also generated at that time from predefined templates.

The way FLORA-2 rules are encoded is more complex. Consider the following rule:

```
Mary[parent->?X] :- Mary[father->?X].
```

This is translated as follows:

```
derived_fd(Mary,parent,?X) :- d_fd(Mary,father,?X).
```

This is done for a number of reasons. The prefix `derived_` is used to separate the head predicates from the body. It is necessary in order to be able to implement inheritance rules correctly, using the XSB well-founded semantics for negation (`not`, see Section 13). The prefix used in the body of a rule, `d_`, is introduced in order to be able to capture undefined methods, *i.e.*, methods those definition was not supplied by the user (see Section 26.1). All these predicates are connected through an elaborate set of rules, which appear in `closure/*.flh` files and also in `genincludes/flrpreddef.flh` (these `flh`-files are generated from the corresponding `fli`-files at FLORA-2 configuration time). The following diagram shows the main predicates involved in the plumbing system that connects the

derived_ and d_; the arrow <---indicates the immediate dependency relationship, *i.e.*, that the predicate on the right appears in the body of a rule that defines the predicate on the left.

In program rules:

```
derived_mvd <--- d_mvd
```

In auxiliary runtime libraries:

```
d_mvd  <---  mvd
d_imvd <--- imvd
          mvd <---      inferred_mvd
          mvd <--- not inferred_mvd      <--- derived_mvd
          mvd <--- not conflict_obj_imvd                     <--- imvddef <--- mvd
          mvd <---      imvd
          mvd <---      immediate_isa

         imvd <---      immediate_sub
         imvd <---      inferred_imvd     <--- derived_imvd
         imvd <--- not inferred_imvd
         imvd <--- not conflict_imvd                         <--- imvddef

      imvddef <---      imvd

  derived_mvd <--- storage_find_fact(...trie_name..., mvd(...))
```

Here we listed only the predicates that are used to model value-returning inheritable (imvd) and non-inheritable (mvd) methods. A similar diagram exists for method signatures. There is additional machinery for IS-A and subclass relationships, and for equality maintenance.

The closure axioms tie all these predicates together to implement the semantics of F-logic. In particular, they take care of the following features:

- Computing the transitive closure of "::" (the subclass relationship). A runtime check warns about cycles in the subclass hierarchy.
- Computing the closure of ":" with respect to "::", i.e., if $X:C, C::D$ then $X:D$.
- Performing monotonic and non-monotonic inheritance. The predicates conflict_obj_imvd, conflict_imvd, immediate_sub, immediate_isa, are used for this purpose.
- Making sure that when the equality maintenance mode changes as a result of the executable instruction :- equality {basic|flogic|none}, program clauses are not overwritten by the rules specified in FLORA-2 runtime libraries. This is the reason for having the wrappers derived_mvd and inferred_mvd. The former appear only in the rule heads of the clauses generated by the compiler from the clauses in the user's program, while the latter appear only in the rule heads in runtime libraries.

- Providing the infrastructure for capturing undefined methods. The purpose of the `d_mvd/mvd` dichotomy is to provide a gap into which we inject rules (which are enabled only if runtime debugging is turned on using `Method[%_mustDefine(on)]`) that can capture calls to undefined methods at run time.

Files that implement these axioms reside in the subdirectory `closure/` and have the suffix `.fli`. These files are used as components from which FLORA-2 *trailers* are created. Trailers are called so because they are typically included at the end of the compiled program. The template for all trailers is found in `includes/flrtrailer.flh`. Several kinds of trailers can be generated from this file: the no-equality trailer (whose main component is `closure/flrnoeqltrailer.fli`), which maintains no equality, and the basic trailer (`closure/flreqltrailer.fli`), which maintains only the standard equality axioms. There are variations of these trailers that also support F-logic inheritance (`flreqltrailer_inh.fli` and `flrnoeqltrailer_inh.fli`).

When a FLORA-2 program is compiled, the compiler includes the trailers into the `.P` file. However, there also is a need to be able to load the trailers dynamically. First, this is needed in the system shell, because the shell is not represented by any particular user program and so there is no place where we can include the trailer. Second, the user might enter the executable instruction

```
?- setsemantics{equality(...)}
```

at the shell prompt and user programs can contain these instructions as part of their code. When an equality maintenance instruction is executed for a particular module, the trailer for that module must be compiled and loaded dynamically. (The need for this compilation will become clear after we explain the implementation of the module system.) These trailers are stored in the user home directory in the subdirectory `.xsb/flora/`. As mentioned earlier, FLORA-2 uses different names for the wrapper predicates that appear in the rule heads in user programs and those that appear in the rule heads in trailers. This makes it possible to load the trailers (by executing the `equality` instruction) at any time without overriding the user program.

The above is a much simplified picture of the inner-workings of FLORA-2. The actual translation into Prolog and the form of the closure rules is very complex. Some of this complexity exists to ensure good performance. Other complications come from the need to provide a module system and integrate it with the underlying Prolog engine. The module system serves two purposes. First, it promotes modular design for FLORA-2 programs, making it possible to split the code into separate files and import objects defined in other modules. Second, it allows FLORA-2 programs to communicate with Prolog by using the predicates defined in Prolog programs and letting Prolog programs use FLORA-2 objects. Some of these implementation issues are described in [11].

The module system. The module system is implemented by providing separate namespaces for the various predicates used to encode F-logic formulas. First, all predicates have a "weird" prefix to make clashes with other Prolog programs unlikely. The prefix is defined in `includes/header.flh` and currently is `_$_$_flora`. The user, of course, does not need to worry about it, unless she runs FLORA-2 programs in a very unfriendly Prolog environment in which other programs also use this

prefix. In this case, the prefix can be made even harder to match.[13]

Apart from the general prefix, each predicate name's prefix contains the module name where this predicate is defined. Since the same F-logic program can be loaded into different modules, the FLORA-2 compiler does not actually know the real names of the predicates it is producing. Instead, it dumps code where each predicate is wrapped with a preprocessor macro. For instance, the predicate `mvd` would be dumped as

```
FLORA_THIS_WORKSPACE(FLORA_USER_WORKSPACE,mvd)
```

where `FLORA_THIS_WORKSPACE` and `FLORA_USER_WORKSPACE` are preprocessor macros. When a program, `myprog.P`, which is compiled by the FLORA-2 compiler, needs to be loaded into a user module, say `main`, the preprocessor, `gpp`, is called with the macro `FLORA_USER_WORKSPACE` set to `main`. Gpp replaces all macros with the actual values. For instance, the above macro expression will be replaced with something like

```
_$_$_flora'usermod'main'mvd
```

Gpp then includes all the necessary files, and then pipes the result to the Prolog compiler. The latter produces the object `myprog.xwam` file where all the predicate names are wrapped with the user module name, as described above. This object file is renamed to `myprog_main.xwam`. If later we need to compile `myprog.P` for another user module, `foo`, `gpp` is called again, but this time it sets `FLORA_USER_WORKSPACE` to `foo`. When Prolog finally compiles the program into the object file, the file is renames to `myprog_foo.xwam`.

It is important to keep in mind that only the predicate names are wrapped with the `FLORA_PREFIX` macro and a module name. Predicate arguments are not wrapped and thus, the space of object Ids is shared among modules. However, this is not a problem and, actually, is very convenient: we can easily refer to objects defined in other modules and yet the same object can have completely different sets of properties in each separate module. This does not preclude the possibility of encapsulating objects, because only the methods need to be encapsulated — oids do not carry any meaning by themselves.

To provide encapsulation for HiLog predicates, they are also prepended with the module name. In particular, this implies that HiLog atomic formulas have different representation than HiLog terms: a formula `p(a,f(b))` would be encoded as

```
FLORA_THIS_WORKSPACE(FLORA_USER_WORKSPACE,apply)(p,a,FLORA_PREFIX'apply(f,b))
```

The same term would be encoded differently if it occurs as an argument of a predicate of another functor:

```
FLORA_PREFIX'apply(p,a,FLORA_PREFIX'apply(f,b))
```

Thus, FLORA-2 implements a 2-sorted version of HiLog [3].

[13] It is necessary to ensure that the resulting predicate names are symbol strings acceptable to the Prolog compiler. Look at the macros `FLORA_THIS_WORKSPACE` and `FLORA_THIS_FDB_STORAGE` in `includes/flrheader.flh` to see what is involved.

The updatable part of the database. All objects and facts that are explicitly inserted by the program are kept in the special *storage trie* associated with the user module where the program is loaded. A tries is a special data structure, which is well-suited for indexing tree-structured objects, like Prolog terms. This workhorse does much of the grudge work in the Prolog engine. To manipulate the storage tries, FLORA-2 uses the XSB package called `storage.P`, which is described in the XSB manual. This package was originally created to support FLORA-2, but it has independent uses as well.

All primitives in this package take a Prolog symbol, called a *triehandle*, a Prolog term, and some also return status in the third argument. Here are some of the most relevant predicates:

```
storage_insert_fact(Triehandle,Term,Status)
storage_delete_fact(Triehandle,Term,Status)
storage_insert_fact_bt(Triehandle,Term,Status)
storage_delete_fact_bt(Triehandle,Term,Status)
```

The first two methods insert and delete in a non-transactional manner, while the last two are transactional.

FLORA-2 associates a separate triehandle (and, thus, a separate trie) with each module. The mechanism is similar to that used for predicate names:

```
FLORA_THIS_FDB_STORAGE(FLORA_USER_WORKSPACE)
```

As explained earlier, when Prolog compiles the file generated by the FLORA-2 compiler, the macro `FLORA_USER_WORKSPACE` gets replaced with the module name and out comes a unique, hard to replicate triehandle.

Unfortunately, putting something in a trie does not mean that Prolog will find it there automatically. That is, if you insert `p(a)` in a trie, it does not mean that the query `?- p(a)` will evaluate to true, and this is another major source of complexity that the FLORA-2 compiler has to deal with. To find out if a term exists in a trie, we must use the primitive

```
storage_find_fact(Triehandle,Term)
```

If the term exists in the trie identified by its triehandle, then the predicate succeeds; if the term does not exist, then it fails. The above primitive can be used to query tries in a more general way, with the second variable unbound. In this case, we can backtrack through all the terms that exist in the trie.

Suppose we insert a fact, `a[m->v]`, represented by the formula `mvd(a,m,v)`. Since this formula is inserted in the trie and Prolog knows nothing about it, we need to connect the trie to Prolog through a rule like this:

```
mvd(O,M,V) :- storage_find_fact(triehandle,f(O,M,V)).
```

Of course, the name of the triehandle and the predicate names must be generated using the macros, as described above, so that they could be used for any module. In FLORA-2 such rules are called *patch rules*.

Since F-logic uses only about the predicates that represent F-molecules, we can create such rules statically and let `gpp` wrap them with the appropriate prefixes on the fly. The problem arises with predicates, since although they are represented using HiLog encoding using a single predicate, this predicate can have any arity. At present we statically create patch rules for such predicates up to a certain large arity. The static patch rules are located in `genincludes/flrpatch.fli` (from which `flrpatch.flh` is generated by the FLORA-2 installation script).

For compiled programs, the patch rules are included into the compiled code by the FLORA-2 compiler. For the FLORA-2 shell, however, these rules are loaded when the corresponding shell module is created (either the default "main" module or any module that was created by the `newmodule` command. This patch file is loaded exactly once per shell module and is kept in the file `.xsb/flora/patch.P`, in the user's home directory.

D.2 System Architecture

The overall architecture of FLORA-2 is depicted in Figure 2. The program is first tokenized and then the *composer* combines the disparate tokens into terms. Since, due to the existence of operators, not everything looks like a term in the source program, the composer consults the operator definitions in the file `flroperator.P` to get the directives on how to turn the operator expressions into terms. Next, the parser checks the syntax of the rules and of the various other primitives (*e.g.*, the aggregates, updates, module specifications, etc.). The output of the parser is a *canonic term* list, which represents the entire parsed program. The canonic term is taken up by the intermediate code generator, which generates abstract code. This code is represented in a form that is convenient for manipulation and is not yet Prolog code. The compiler might add additional rules (such as patch rules) and Prolog instructions. The compiled program is converted into (almost) Prolog syntax by the coder. As mentioned previously, the code produced by the compiler is full of preprocessor macros, so before passing it to Prolog it must be preprocessed by GPP. GPP pipes the result to Prolog, which finally produces the byte code program that can run under the control of the Prolog emulator.

The following is a list of the key files of the system.

- `flrshell.P`: The top level module that implements the FLORA-2 shell — a subsystem for accepting user commands and queries and passing them to the compiler. See Section 2 for a full description of shell commands.
- `flrlexer.P`: The FLORA-2 tokenizer.
- `flrcomposer.P`: The FLORA-2 composer, which parses tokens according to the operator grammar and does other magic.
- `flrparser.P`: The FLORA-2 parser.
- `flrcompiler.P`: The generator of the intermediate code.
- `flrcoder.P`: The FLORA-2 coder, which generates Prolog code.

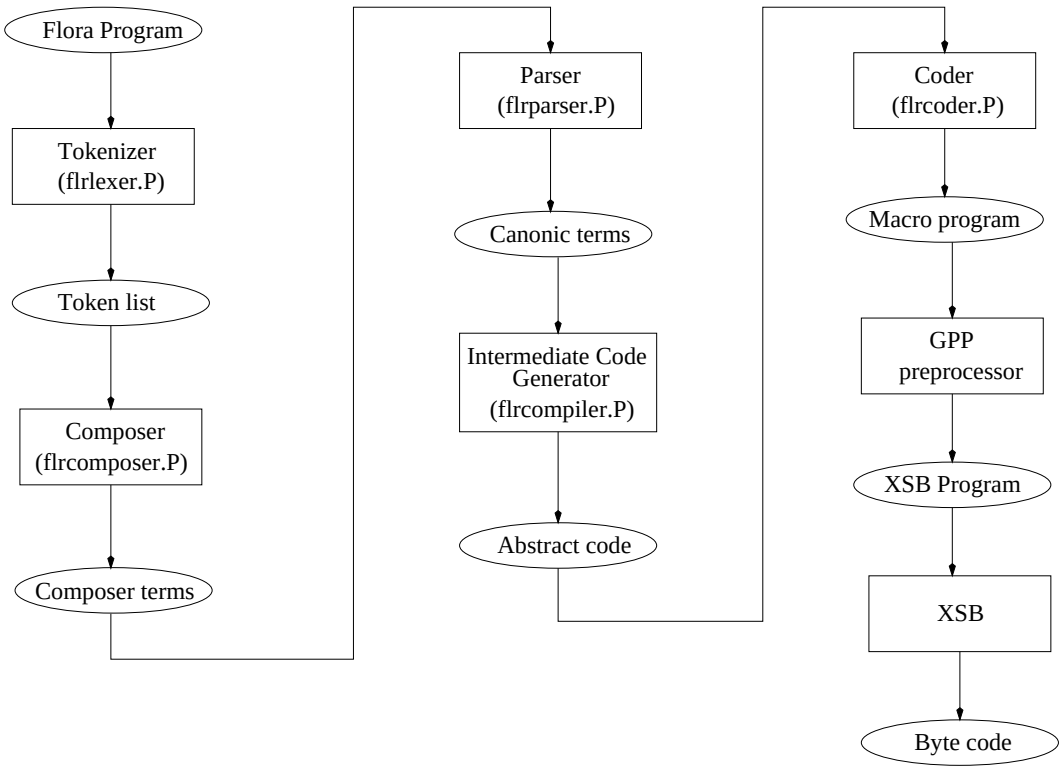

Figure 2: The architecture of the FLORA-2 system.

- `flrutils.P`: Miscellaneous utility predicates for loading programs, checking if files exist, whether they need to be recompiled, etc.

Additional system libraries are located in the `syslib/` subdirectory. These include the various printing utilities, implementation for aggregates, update primitives, and some others. The compiler determines which of these libraries are needed while parsing the program. When a library is needed, the compiler generates an `#include` statement to include an appropriate file in the `syslibinc` directory. For instance, to include support for the `avg` aggregate function, the compiler copies the file `syslibinc/flraggavg_inc.flh` to the output `.P` file. Since `syslibinc/flraggavg_inc.flh` contains the code to load the library `syslib/flraggavg.P`, this library will be loaded together with that output file. The association between the libraries and the files that need to be included to provide the appropriate functionality is implemented in the file `flrlibman.P`, which also implements the utility used to load the libraries.

While `syslib/` directory contains the libraries implemented in Prolog, the `lib/` directory contains libraries implemented in FLORA-2 itself. Apart from that, the two types of libraries differ in functionality. The libraries in `syslib/` implement the primitives that are part of the syntax of the FLORA-2 language itself. In contrast, the libraries in `lib/` are utilities that are part of the system, but not part of the syntax. An example is the pretty-printing library. Methods and predicates defined in the libraries in `lib/` are accessible through the `@_libname` system module and (unlike

user modules) they are loaded automatically at startup.

There are several subdirectories that hold the various files that contain definitions included at compile time. These will be described in a technical document.

A number of other important directories contain the various included files (many of which include other files). The directory `flrincludes/` contains the all-important `flora_terms.flh` file, which defines all the names used in the system. These names are defined as preprocessor macros, so that it would be easy to change them, if necessary. The directory `genincludes/` currently contains the already mentioned patch rules. The file `flrpatch.fli` is a template, and `flrpatch.flh`, which contains the actual patch rules, is generated from `flrpatch.fli` during the installation.

The directory `includes/` contains (among others) the header file, which defines the a number of important macros (*e.g.*, `FLORA_THIS_WORKSPACE`) that wrap all the names with prefixes to separate the different modules of the user program. The directory `headerinc/` is another place where the template files are located. Each of these files contains just a few `#include` statements, mostly for the files in the `closure/` directory (which, if you recall, contains pieces of the trailer). All meaningful combinations of these pieces of the trailer are represented in the file `includes/flrtrailer.flh`. (Recall that trailers implement the closure axioms.)

The directory `p2h` contains (the only!) C program in the system. It implements conversion of Prolog terms to HiLog and back. Finally, the `pkgs/` directory is empty. Some day it will contain add-on programs, such as Internet access, etc.

References

[1] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[2] 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.

[3] W. Chen and M. Kifer. Sorted HiLog: Sorts in higher-order logic programming. In *Int'l Conference on Database Theory*, number 893 in Lecture Notes in Computer Science, January 1995.

[4] 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.

[5] K. Clark. Negation as failure. *Logic and Databases*, pages 293–322, 1978.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] G. Yang and M. Kifer. Implementing an efficient DOOD system using a tabling logic engine. In *First International Conference on Computational Logic, DOOD-2000 Stream*, July 2000.

[12] G. Yang and M. Kifer. On the semantics of anonymous identity and reification. In *Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE)*, October 2002.

[13] G. Yang and M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE)*, October 2002.

[14] G. Yang and M. Kifer. Inheritance and rules in object-oriented semantic web languages. In *Rules and Rule Markup Languages for the Semantic Web (RuleML03)*, number 2876 in Springer Verlag, October 2003.

Index