$\mathcal{E}$RGO^{Lite} (a.k.a. $\mathcal{F}$LORA-*2*): User's Manual

Version 2.0

(Pyrus nivalis)

Michael Kifer Guizhen Yang Hui Wan Chang Zhao

with contributions from
Polina Kuznetsova and Senlin Liang

Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400, U.S.A.

October 13, 2018

Contents

1 Introduction

$\mathcal{E}$RGO^{Lite} (also known under its older name $\mathcal{F}$LORA-*2*) is a sophisticated object-based knowledge representation and reasoning system. $\mathcal{E}$RGO^{Lite} is an open source subset of its commercial cousin called *$\mathcal{ERGO}$ Reasoner*,[1] which contains many important extensions and enhancements to $\mathcal{F}$LORA-*2*, and is proprietary to Coherent Knowledge Systems. $\mathcal{F}$LORA-*2* is implemented as a set of run-time libraries and a compiler that translates a unified language of F-logic [10], HiLog [5], Transaction Logic [3, 2], and defeasible reasoning [16] into tabled Prolog code.

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

The language of $\mathcal{E}$RGO^{Lite}/$\mathcal{F}$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, logical updates in the style of Transaction Logic, and a form of *defeasible reasoning* described in [16]. $\mathcal{F}$LORA-*2* was designed with extensibility and flexibility in mind, and it provides strong support for modular software development 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.[2] 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 [10], and from an earlier implementation, $\mathcal{F}$LORA-*1*. These syntax changes made the system more user-friendly and practical.

$\mathcal{E}$RGO^{Lite}/$\mathcal{F}$LORA-*2* is available at `http://flora.sourceforge.net`. This manual will mostly refer to the system under its traditional name, $\mathcal{F}$LORA-*2*.

2 Installation

$\mathcal{F}$LORA-*2* has a *simplified* installation procedure for the official, pre-compiled releases of the system[3] and a full procedure for those who downloaded the very latest development version of the software directly from the SVN repository.[4] Most users would use the simplified installation procedure, especially on Windows.

[1] `http://coherentknowledge.com`
[2] `http://www.informatik.uni-freiburg.de/~dbis/florid/`.
[3] `https://sourceforge.net/projects/flora/files/FLORA-2/`
[4] For online browsing: `https://sourceforge.net/p/flora/src/HEAD/tree/`.
For downloading: `svn checkout svn://svn.code.sf.net/p/flora/src/trunk flora-src`.

2.1 Simplified Installation

You can use this procedure, if you downloaded an official pre-compiled release of *F*LORA-*2*; it *cannot* be used if you obtained the software from the SVN repository. The advantage of this procedure is that everything installs with just one command and on Windows it does not require any special software, such as the C++ compiler or nmake.

The official release also includes one of the latest versions of XSB that is compatible with the *F*LORA-*2* release. Thus, there is no need to download XSB separately.

Unix-based systems (Linux, Mac, BSD, etc.). The Unix-style download is the file `flora2.run`. This is a self-extracting archive. Put it in a suitable installation directory, **DIR** and then execute

```
cd DIR
sh ./flora2.run
```

This will create the folder `Flora-2` with two subfolders: `flora2` and `XSB`. The script to run *F*LORA-*2* is `DIR/Flora-2/flora2/runflora`.

Windows. The Windows installer is called `flora2.exe`. Put it on your desktop and double-click on it. You will be asked a series of questions, as usual with Windows installers. In the end, both XSB and *F*LORA-*2* will be installed with the appropriate shortcuts placed on the desktop and in the Start menu.

For 64-bit machines, the 64-bit version will be installed automatically. Otherwise, the 32-bit version will be installed.

Starting *F*LORA-*2*. The command to run *F*LORA-*2* is `./runflora` (Linux/Mac/BSD/etc.) or `runflora.bat` (Windows). These scripts are located in the `flora2` subfolder of the installation folder. In most cases, the installer will place a *F*LORA-*2* icon on your desktop, and you can run the software by double-clicking on that icon.

You may notice that during the first couple of uses of the system, the system startup and some queries are delayed by 2-4 seconds. This is normal and is due to the compilation of some libraries. The delay will disappear after a few uses.

2.2 Installing *F*LORA-*2* from SVN

This procedure is significantly more complex than the one for installing the official *F*LORA-*2* releases described above. It is suitable only for the users who need the very latest development version of

FLORA-2—either in order to be up-to-date with respect to the bug fixes or in order to try the new cutting-edge features.

First, install the most recent version of the XSB engine from XSB's SVN repository.[5]

XSB sources must be installed. If you used the XSB Windows installer, make sure that the appropriate boxes are checked. Otherwise, the installer will install only the basic system and the subsequent installation of FLORA-2 might fail.

Compile XSB, as explained in XSB's manual, Section 2.2.2. It is highly recommended that you compile XSB as a 64-bit application (if your hardware is 64 bit). Note that XSB *cannot* be compiled in a directory whose path contains spaces. However, once compiled, XSB can be moved to another directory, which may contain spaces. No recompilation is necessary after the move.

Next, check out FLORA-2 from its SVN repository[6] into a separate directory *outside* the XSB installation tree. The FLORA-2 sources will be placed in the `flora2` subdirectory of the main development branch.

Make sure that the typical development software is installed on your machine. On Unix-based systems, this includes the GCC compiler, Make, and Autoconf. On Windows, this means Microsoft Visual Studio, Community Edition (a free download) or a full (paid) Studio version.

Linux, Mac, BSD, etc. Configure FLORA-2 as follows:

```
cd flora2
make clean
./makeflora all path-to-/XSB/bin/xsb
```

If the XSB executable is on your program search `PATH`, then instead of the third command above you can simply type `./makeflora`.

Windows. You need Microsoft's Visual Studio with the C++ component installed. The free of charge Microsoft Visual Studio, Community Edition, can be downloaded from

```
https://www.visualstudio.com/vs/community/
```

Next, find the `nmake.exe` program within Visual Studio and put the folder it is in on the `PATH` environment variable. In fact, you really only need this `nmake.exe`, not the entire studio, unless you are planning to add components written in C.

[5] For browsing: https://sourceforge.net/p/xsb/src/HEAD/tree/.
For downloading: `svn checkout svn://svn.code.sf.net/p/xsb/src/trunk xsb-src`.

[6] To download, use: `svn checkout svn://svn.code.sf.net/p/flora/src/trunk flora-src`

Then configure FLORA-2 as follows:

```
cd flora2
makeflora clean
makeflora path-to-\XSB\bin\xsb64.bat
```

Here `flora2` is the root directory for FLORA-2 in your SVN checkout; it should have the form *something*\`flora2`. The argument *path-for*-`\XSB\bin\xsb.bat` must be the full path name for the XSB invocation script. Note, that unlike Linux/Mac, there should be no "all" keyword in the Windows version of the `makeflora` command.

If for some reason you compiled XSB as a 32-bit application, use `xsb.bat` above instead of `xsb64.bat` in the makeflora command.

If you are a developer and wish to recompile the C part of FLORA-2 (say, because you changed it to fix a bug or added a feature) then you can type

```
makeflora -c64 path-for-\XSB\bin\xsb64.bat
```

(`-c` and `xsb.bat` for 32-bit versions of XSB). For this, the C++ component of Visual Studio must be installed. Normally, however, there is no need to perform this step. For "`makeflore -c64`" to work, it may be necessary to locate the file `vcvars64.bat` or `vcvars32.bat` — depending on whether your XSB is 64 or 32 bits — and execute that batch file in the command window where `makeflora -c` is to be run.

Note: FLORA-2 does not work with Cygwin due to a problem with XSB calling external C procedures under Cygwin.

3 Running FLORA-2

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

The easiest way to get a feel of the system is to start the FLORA-2 shell and enter some queries interactively:

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

Here "..." stands for the directory in which FLORA-2 is installed. For instance,

```
~/ENGINEDIR/flora2/runflora
```

At this point, FLORA-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 the FLORA-2 shell frequently, it pays to define an alias, say (in Bash):

```
alias flora2='~/ENGINEDIR/flora2/runflora'
```

FLORA-2 can then be invoked directly from the shell prompt by typing `flora2`. It is even possible to tell FLORA-2 to execute commands on start-up. For instance, typing

```
flora2 -e "\help."
```

in the command window of your operating system will cause the system to execute the help command right after after the initialization. Then the usual FLORA-2 shell prompt is displayed.

FLORA-2 comes with a number of demos that live in

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

The demos can be run with 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 examples.

The initialization file. When FLORA-2 starts up, it first executes the commands found in the initialization file, if it is specified and exists. The initialization file is specified as a value of the `FLORA_RC_FILE` environment variable of the operating system in use. If this variable is not set or if the value of that variable is not a readable plain file, the initialization file is ignored. The commands in the initialization file can be any kind of FLORA-2 queries or commands. They must be specified exactly as they would be written in the interactive FLORA-2 shell, i.e., without the query prefix "`?-`" and they must be terminated with the period. For instance, if `FLORA_RC_FILE` is set to `~/test/myrc` (note: the `.flr` suffix is not required) and that file contains

```
writeln('Welcome!')@\plg.
insert{foo(bar)}.
```

then the message "Welcome!" will be printed and the fact `foo(bar)` will be inserted into the knowledge base.

There is one restrictions on the initialization file: a comment cannot be the last statement. Also note that the command-line option `-e` mentioned earlier is executed *after* all the initialization file commands are processed.

4 FLORA-2 Shell Commands

The FLORA-2 shell (or command line interface) is designed to do what its name says: to accept commands that the user wishes FLORA-2 to execute. These commands can roughly be divided into these categories:

1. *Loading*: commands for loading knowledge base files into FLORA-2, e.g., `load{...}`, `add{...}`.
2. *System information and control*: various commands that provide information about the state of the system, e.g., `system{...}`, `semantics{...}`, `isloaded{...}`, `\one`, `\trace`.
3. *Execution control*: In ERGO, one can also monitor, examine, stop, and resume execution of running queries, e.g., `setmonitor{...}`.
4. *Knowledge base querying*: ask queries about the user knowledge base, e.g., `Ann[address->?A]` or `?p:Person[phone->?f], fmt_write('%s\'s phone is %s\n', args(?p,?f))@\io.` (What is Ann's address? Find all people and their phones and then print out the result as a block of lines of the form *...'s phone is ....*)

These commands all have the form of a *rule body* — see Section 8.1 for the exact syntax of rule bodies.

Loading knowledge bases from files. The most common shell commands you probably need are the commands for loading and compiling knowledge bases:

```
flora2 ?-   [myfile].        // e.g., ['c:/My Documents/data'].
flora2 ?-   [url(myurl)].    // e.g., [url('http://example.com/data')].
```

or

```
flora2 ?-  load{myfile}.      // e.g., load{'/home/me/proj/kb'}.
flora2 ?-  load{url(myurl)}.
```

Here *myfile* or *myurl* are a file names (respectively, a URL) that are assumed to be pointing to an FLORA-2 knowledge base or a Prolog program. Both *myfile* and URL must be Prolog atoms. If they contain non-alphanumeric characters (as in the examples above) then they must be single-quoted (as usual for Prolog atoms). A URL is expected when the argument has the form `url(`*`myurl`*`)`.[7] The file can be relative to the directory in which FLORA-2 was started. For instance, if that directory has the file `foo.flr` then one can simply type `[foo]` instead of `['/home/me/foo.flr']` or even `['/home/me/foo']`. Note that our first example file, `'c:/My Documents/data'`, is a Windows file name, except that it uses *forward* slashes (like in URLs), which is preferred to backward slashes. Backward slashes can also be used in Windows, but they must be doubled: `'c:\\My Documents\data'`. On Unix-based systems, such as Linux and Mac, only forward slashes can be used.

If *`myfile`*`.flr` exists, it is assumed to be a knowledge base written for FLORA-2. The system will compile the knowledge base, if necessary, and then load it. The compilation process is two-stage: first, the knowledge base is compiled into a Prolog program (one or more files with extensions `.pl`, `.fdb`, and others) and then into an executable byte-code, which has the extension `.xwam`. For instance,

```
flora2 ?-  load{url('http://example.com/test1')}.
flora2 ?-  [url('http://example.com/test2')].
```

will compile (if necessary) and load the FLORA-2 files `test1.flr` and `test2.flr` found at the Web site `http://example.com/`.

If there is no *`myfile`*`.flr` file, the file is assumed to contain a Prolog program and the system will look for the file named *`myfile`*`.P`. This file then is compiled into *`myfile`*`.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 17.1 for the details about the module system in FLORA-2.

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

```
flora2 ?-   [myfile>>modulename].
flora2 ?-   [url(myurl)>>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 17.1. Once the knowledge

[7] For this to work, the XSB package `curl` must be configured as described in the XSB manual, volume 2.

base is loaded, you can pose queries and invoke methods for the objects defined in that knowledge base.

All the loading commands that apply to files also apply to URLs, so in the future we will be giving examples for files only.

There is an important special case of the `load{...}` and `[...]` commands when the file name is missing. In that case, FLORA-2 creates a scratchpad file and starts reading user input. At this point, the user can start typing in FLORA-2 clauses, which the system saves in a scratchpad 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 the scratchpad file into a designated module, rather than the default one, using one of the following commands:

```
flora2 ?-  [>>module].
flora2 ?-  load{>>module}.
```

Adding rulebases 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 rulebase 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.

Reloading and re-adding. Recompilation. FLORA-2's `load{...}` and `add{...}` commands try to be smart in order to simplify maintenance of knowledge bases and to avoid undesirable side effects. First, reloading and re-adding the same file to the same module will have no effect unless

one of the *dependent* files has changed since the previous load/add. So, cyclic add/load commands are harmless, albeit they constitute evidence of bad design.

For the purpose of recompilation, a dependent file is one that is included with the `#include` compiler directive. The dependent property is transitive, so if any of the dependent files down-stream from the parent changes, loading or adding the parent file will cause that parent to be recompiled.

Similar relationship exists with respect to the load/add dependency. Normally, as we said, reloading a file will have no effect. But what if the file being loaded (or added) has an explicit load/add command that loads another file (which we call *load-dependent*)? If the dependent file was changed since the last loading, it needs to be reloaded and recompiled. In this case, if the parent file is reloaded then this reloading *will* take place and so all the load/add commands in that file will be re-executed causing the reloading of all the relevant load-dependent files. Such loading will take place if any of the load-dependents changes—at any level down-stream from the parent.

Reporting query answers. 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 their number is finite). 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 output control command `\one` will affect `goallist2`, but *not* `goallist1`. This is because `goallist1` executes in the same statement as `\one` and thus is not affected by this directive.

The 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.

System information. The `system{...}` command can be used to obtain information about the current version of FLORA-2 and the underlying OS. For instance,

```
flora2 ?-  system{system=?X}.
?X = Ergo
```

```
flora2 ?-  system{version=?X}.
?X = '1.3'
flora2 ?-  system{info=?X}.
?X = 'Ergo 1.3 (Zeno; build: aa9225b, 2018-01-27) on x86_64-unknown-linux-gnu'
```

To see all the available system information and all the valid properties that can be used in the `system{...}` command, ask the query `system{?X}`.

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 knowledge base, 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 knowledge base 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.

- `\help`: Show the help info.
- `system{param}`: Show system information.
- `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 it will load those 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 to 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.
- `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` *and* `\notrace`: Turn on/off FLORA-2 trace.
- `chatter{on}` *and* `chatter{off}`: Turn on/off the display of the number of solutions at the end of each query evaluation.
- `feedback{on}` *and* `feedback{off}`: Turn on/off the display of query answers. Mostly used in Java applications.
- `setwarnings{type}`: Control the types of warnings to be shown to the user.
  - `all` — show *all* warnings (default)
  - `off` — do not show *any* warnings
  - `compiler=on`, `compiler=off` — turn compiler warnings on (default) or off; no effect on other types of warnings
  - `dependency=on/off` — turn dependency checker warnings on (default) or off; does not affect other types of warnings
  - `runtime=on/off` — turn runtime warnings on (default) or off; does not affect other types of warnings.
- `warnings{?Type}`: tell which warning control options are in effect. `?Type` can be a variable or a pattern like `compiler=?X`.
- `\end`: Say Ciao to FLORA-2, but stay in Prolog. You can still re-enter FLORA-2 by executing the `flora_shell` command at the Prolog prompt.
- `\halt`: Quit both FLORA-2 and Prolog.

Of course, many other executable directives and queries can be executed at the 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 17.9. 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

```
++Warning[Flora-2]: 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 ?-
```

5 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 available on the FLORA-2 Web site: http://flora.sourceforge.net/tutorial.html.

6 Differences Between FLORA-2 and the F-logic Syntax

FLORA-2 was developed years after the publication of the initial works on F-logic [10] and so it benefits from the experience gained in the use and implementation of the logic. This experience suggested a number of 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-logic frame formulas. The version of the logic in [10] 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 $\mathcal{F}$LORA-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 [10], types are always inheritable, but values are not. In $\mathcal{F}$LORA-2, information about an object is strictly separated into information about the object proper and information about its superclasses. The latter information is *inherited* by the objects. The original F-logic in [10] used four types of arrows: `->`, `->>`, `=>` (and `=>>` because it distinguished between functional and set-valued methods), and both were inheritable. $\mathcal{F}$LORA-2 uses only two types of arrows: `->` for values and `=>` for types. However, there is now a new type of a formula that employs these arrows: `obj[|meth->val|]` and `obj[|meth=>type|]`. Here, `obj` is considered as a class and `obj[|meth->val|]` specifies the default value of the method `meth`, which is inherited by the subclasses and members of `obj`, while `obj[|meth=>type|]` specifies the type of `meth`, which is also inherited by the subclasses and members of the class `obj`.

   The semantics of this new type of formulas can be characterized by the following logical entailments ($\phi \models \psi$ means ϕ logically entails ψ):

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

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

6. Equality (the `:=:` predicate) is implemented only partially in $\mathcal{F}$LORA-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 23.1.

7. Behavioral inheritance has a different (and better) semantics in $\mathcal{F}$LORA-2 compared to [10]. This is discussed in Section 22.

8. $\mathcal{F}$LORA-2 also has many extensions compared to F-logic. First, it supports HiLog [5, 4] and Transaction Logic [1, 2, 3]. Second, it supports a form of defeasible reasoning known as LPDA (logic programs with argumentation theories) [16]. Third, $\mathcal{F}$LORA-2 has many syntactic extensions, including full first-order formulas that can appear in the rule bodies, many useful builtins, set notation, etc.

7 Main Syntactic Elements of $\mathcal{F}$LORA-2

$\mathcal{F}$LORA-2 has rich syntax, so it is useful to first list the various types of statements one may encounter in this manual. First, we should note that $\mathcal{F}$LORA-2 does *not* have any alphanumeric *reserved* keywords, so the user is not restricted in that name space in any way. The only reserved keywords are those that start with a backslash, e.g., `\and`, `\or`, `\if`, etc.

The main types of FLORA-2 statements are compiler and runtime directives, rules, queries, latent queries, and facts. The basic syntax of these statements is described in Section 8.1 and additional features are introduced in subsequent sections.

- **Compiler directives** have the form

  ```
  :- directiveName{arguments}.
  ```

  Some directives do not have arguments. Compiler directives affect the compilation of the file in which they appear—typically the semantic and optimization options.

- **Runtime directives** have the form

  ```
  ?- directiveName{arguments}.
  ```

  Runtime directives are typically used to change the semantics of the runtime environment at run time.

- **Rules** have the form

  ```
  @!{statementDescriptors}  ruleHead  :-  ruleBody.
  ```

  Rules constitute the key part of an FLORA-2 knowledge base as they (along with the class hierarchy) represent the actual *knowledge*. The presence of rules is the main difference between knowledge bases and mere databases.

  The statement descriptor part (`@!{...}`) is optional. The body of a rule is sometimes also called a *premise*.

  Rules usually appear in files and are added to the system when these files are loaded or added to modules. Less often, they may also be worked with in the FLORA-2 shell. For instance, they can be inserted via the statements `insert{...}`, `insertrule{...}`, deleted (via `delete{...}`, etc.), enabled, disabled (`enable{...}`, `disable{...}`), and queried (via `clause{...}`). See Sections 27, 29, and 36.2. All these commands can be used not just in the shell but also in the bodies of rules and in queries that appear in files.

- **Queries** have the form

  ```
  ?- ruleBody.
  ```

  Syntactically queries have the same form as the rule bodies, but they use the symbol "`?-`" to distinguish the two. Queries are used to request information from the knowledge base.

  Note: the prefix "`?-`" in front of a query is used *only* if the query is in a file. In the FLORA-2 shell, this prefix is *not* used (and will cause a syntax error) because the shell expects queries only—typing rules, directives and other constructs will cause errors to be issued.

- **Latent queries** have the form

  ```
  @!{statementDescriptors}  !- ruleBody.
  ```

  Latent queries are similar to regular queries. However, regular queries are *immediate* requests for information from the knowledge base, while latent queries are requests that intended to be posed at a later time. A latent query also has descriptors, which are used to refer to the query and to invoke it.

- **Facts** are statements that are considered to be unconditionally true. They have the form

  ```
  @!{statementDescriptors}  ruleHead.
  ```

  Syntactically and conceptually, a fact is a rule without a premise. The descriptor part of the syntax for facts is optional.

  Note: a simple fact (without the optional descriptor) looks like a query without the "?-" prefix. A novice might make a mistake by typing facts into the FLORA-2 shell, expecting that these facts will be *inserted* into the knowledge base. However, since the shell expects queries only, this will result in these facts being asked as queries and most likely getting the answer `No`. To insert queries (and rules) via the FLORA-2 shell, use the `insert{...}` command—see Section 27.

Rule heads, bodies, queries, and their arguments are typically composed out of *base formulas* with the help of connectives, such as conjunction, disjunction, the various negations, and more. The main forms of base formulas are

- F-logic **frames** are used for object-oriented knowledge representation.
- HiLog **predicates** are used for more traditional knowledge representation. However, in FLORA-2, predicates can be higher-order and variables are allowed to range over them.

The different types of frames and predicates are described in the respective sections. The main components used to construct predicates and frames include:

- **Variables**, which are expressions of the form *?Varname*.
- **Constants**, which includes *symbols*, *strings*, *numbers*, and various other data types. There are certain builtin constants, like `\true`, `\false`, and `\undefined`, which represent the three truth values in FLORA-2: *true*, *false*, and *undefined*.
- **Operators**, including arithmetic operators.

- **Quasi-constants** and **quasi-variables**. *Quasi-constants* are symbols that get substituted with real constants at compile time or at the time the knowledge base is loaded. The substitution depends on the context in which the symbol occurs and this is why such symbols are called *quasi*-constants and not constants. A *quasi-variable* is a symbol that denotes a variable that gets instantiated at run time by the system and provides certain runtime meta-information. Such variables cannot be affected by the user directly, but the user can check their values.

  Examples of quasi-constants are `\@F` and `\@L`, which get substituted with the file name and the line number in which these constants occur. Quasi-constants let one write statements that refer to objects, such as the file name or line number, which are either unknown at the time of writing or may change later. Quasi-constants never change during runtime—it is just that their values are typically unknown at the time these constants are written into the knowledge base by the knowledge engineer.

  As example of a quasi-variable is `\?C`. This variable can appear in the body of a rule and it gets instantiated with the name of the module from which that rule was called. This is a *quasi*-variable because it gets instantiated by the runtime system and is not under the control of the knowledge engineer. It is a *variable* (rather than a constant) because it may get instantiated with different values during runtime (because, different modules may invoke the same rule).

- **Auxiliary symbols**, such as `->`, `!`, `[`, `(`, `[|`, etc., are used to glue together the aforesaid components to form base formulas.

8 Basic FLORA-2 Syntax

In this section we describe the basic syntactic structures used to specify FLORA-2 knowledge base. 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-2 precisely, because it is based on operator grammar (as in Prolog) mixed with context free grammars in places where operator grammar is inadequate (as, for example, in parsing if-then-else).

8.1 F-logic Vocabulary

- *Symbols*: The F-logic alphabet of *object constructors* consists of the sets $\mathcal{C}$ and $\mathcal{V}$ (constants and 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 *general constant symbols* or *Prolog atoms*. General constant symbols can also be any sequence of symbols enclosed in single quotes (e.g., 'AB@*c'). Later, in Section 39, we introduce additional constants, called typed literals.

In addition to the usual first-order connectives and symbols, FLORA-2 has a number of special symbols:], [, }, {, ",", ";", %, #, \#, ->, =>, :, ::, ->->, -->>, :=:, etc.

- *Numeric constants*: These include *integers*, like 123 or 5063; *decimals* of the form 123.45; or *floating point numbers*, like 12.345e12 ($= 12.345 * 10^{12}$), 0.34e+3 (same as 0.34e3), or 360.1e-2 ($= 360.1 * 10^{-2}$).

- *Anonymous and don't care variables*: Variables of the form ?_ or ? are called *anonymous* variable. They are used whenever a *unique* new variable name is needed. In particular, two different occurrences of ?_ or ? in the same clause are treated as *different* variables. Named variables that start with ?_, 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 care 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 [5] 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.

- *Base 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 predicate formulas, like $p(X_1, \ldots, X_n)$, FLORA-2 allows higher-order *HiLog* base formulas of the form ?X(s,?Y), ?X(f,?Y)(?X,g(k)), etc., where ?X and ?Y are variables, while the symbols not prefixed with a ? are constants. Furthermore, the following *frame* formulas are supported in FLORA-2:

1. `O[M->V]`, `C[|M->V|]`
2. `O[M->{`V_1`,...,`V_n`}]`, `C[|M->{`V_1`,...,`V_n`}|]`
3. `O[M=>T]`, `C[|M=>T|]`, `C[|M{L..H}=>T|]`
4. `O[V]`, `C[|V|]`
5. `O[=>T]`, `C[|=>T|]`
6. `O[ ]`, `C[| |]`

Here `O`, `C`, `M`, $\texttt{V}_\texttt{i}$, $\texttt{T}_\texttt{i}$ are *HiLog terms* of the form `a`, `f(?X)`, `?X(s,?Y)`, `?X(f,?Y)(?X,g(k))`, etc., where `?X` and `?Y` are variables and `f`, `s`, etc., are constants.

Expressions (1) and (2) above are *data frames* 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, $\texttt{V}_1$, ..., $\texttt{V}_\texttt{n}$, 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 constraints. The formula (2) says that the result consists of several objects, which *includes* $\texttt{V}_1$, $\texttt{V}_2$, ..., $\texttt{V}_\texttt{n}$. 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.

In (1) and (2), when `M` is a constant, *e.g.*, `abc`, then we say that it is an *attribute* or a *property*; for example, `John123[name ->'John Doe']`. 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, `John123[salary(2017)->50000]`. However, as we saw earlier, method expressions can be much more general than these two possibilities: they can be arbitrary HiLog terms.

The formulas in (1) and (2) that use `[|...|]` apply to *classes* and specify the *default values* inherited by the objects that belong to those classes. To make it easier to remember, we use the letter `C` in those cases. The letter `O` (for "object") is used with formulas of the form `[...]`, which apply to individual objects and specify *concrete* values of their attributes/methods as opposed to the default values inherited from superclasses. These concepts are explained in greater detail in Section 22.

The expression (3) above is a *signature frame* (or *typing frame*). It specifies a *type constraint* that says that the method expression, `M`, when applied to objects that belong to class `C`, must yield objects that belong to class `T`. The first form of type constraints in (3) applies to individual objects, the second to classes, and this second form is inherited by subclasses and individual objects. Third, while also applying to classes, imposes a cardinality constraint on the possible number of values `T` that can correspond to the same `M`. Here `L>0` is a number that specifies the lower bound on that number of values and `H` specifies the upper limit. See Section 22 for additional details on this type of formulas.

Note: FLORA-2 does not automatically enforce type constraints. Also, run-time type checking is possible—see Section 43.2. □

The form (4) is used for Boolean methods. Unlike the methods in (1) and (2), which can return any kind of results, Boolean methods can be either true or false: they do not return any values. Apart from that, the previous conventions apply: `O[V]` says that object `O` has a Boolean property `V` and `C[|V|]` says that class `C` has a *default* Boolean property that is inherited by `C`'s subclasses and member objects.

The form (5) specifies signatures, i.e., types for the formulas in (4). Note that, unlike in (3), cardinality constraints do not apply in this case.

Empty frames. Normally, a frame has at least one statement about the frame's object, as in

```
John[spouse -> Mary]
```

However, as seen from case (6) above, a frame can also be *empty* like this: e.g.,

```
Mary[]
Mary[| |]
```

Here we have an empty object specification and an empty class signature. This is to be interpreted as a statement that the corresponding object (`Mary`, in our case) is *known to exist* in the domain of discourse. This means that some statement about that object is derivable from user specifications. For instance, if the knowledge base has statements that imply, say, `Mary:person`, `Mary[age->25]`, or if the "empty" fact `Mary[]` exists in the knowledge base, then `Mary[]` holds true. This rule excludes builtin datatypes (see Section 39), such as `\integer`, `\object`, and `\symbol`. For instance, even though `2:\integer` and `foobar:\object` are true, by definition, both `2[]` and `foobar[]` would be false unless these facts are implied by other parts of the knowledge base given by the user, i.e., unless the user "told" the knowledge base about the existence of these objects.

Objects are grouped into classes using what we call *ISA-literals*, which account both for class membership and subclass relationships:

5. `O:C`
6. `C::D`

The expression (5) states that `O` is an *instance* of class `C`, while (6) states that `C` is a *subclass* of *D*.

User-defined equality

7. `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 built-in `=`, which exists both in FLORA-2 and Prolog. For instance, `=`-based formulas can never occur as a fact or in a rule head, and `a = b` is always false. More on user-defined equality in Section 23.1.

8. *Composite frames.*

F-logic frames (or *frame literals*) and ISA-literals, can be combined in various ways, reducing long conjunctions into very compact forms. For instance, the conjunction of `John:person`, `Bill:Student`, `John[age->31]`, `John[children->Bob]`, `John[children->Mary]`, `John[children->Bill]`, and `Mary[age->5]` can be compacted into the following complex frame:

```
John:person[age->31, children->{Bob,Mary[age->5],Bill:Student}]
```

Note that this shows that frames can be nested (e.g., `Mary[age->5]`) and values pertaining to the same method and object can be grouped into sets (e.g., `John[children->{Bob,Mary,Bill}]`). Furthermore, ISA-literals can be attached both to the outermost object as well as the inner ones (as in `John:person` and `Bill:Student`).

Furthermore, in rule bodies and queries, frame attribute notation combines with the binary operators `=`, `\=`, `!=`, `==`, `\==`, `=\=`, `=:=`, `!==`, `:=:`, `\is`, `>`, `<`, `=<`, `>=`, `@<`, `@>`, `@=<`, `@>=`, ∼, `!`∼, `\in`, `\subset`, `\sublist`. These operators will be introduced as we go, many in Section 8.5, but here is an example of a query where some of these operators are rolled into a frame:

```
?- John[age->?A>30, children->?C != Bob, salary->?X \is 200*7+5000,
        phone->?P \in ['111-222-3456','123-456-7890']].
```

This is a shorthand for

```
?- John[age->?A, children->?C, salary->?X], ?A>30, ?C!=Bob, ?X \is 200*7+5000.
```

- *Atomic formulas.* The base formulas of the types (1)-(7) above are called *atomic formulas.* Atomic formulas are also base formulas, but the latter can also contain non-atomic formulas: complex frames.
- *Rules* are constructs of the form *head*`:-`*body*, where
  - *head* is a frame/HiLog literal or a conjunction of such literals. These literals can also be negated with `\neg`. and

– *body* is a conjunction and/or disjunction of frame/HiLog literals or negated (with \+, \neg, or \naf) frame/HiLog literals. Each rule must be terminated with a ".".

Conjunction is specified, as in Prolog, using the "," symbol but \and is also accepted. Disjunction is denoted using the semicolon symbol ";" or using \or. Negation is specified using \+, \neg, or \naf— the difference will be explained later. For example,

```
p(?X), \neg ?Y[foo->bar(?X)] :- (q(?X,?Y) \or ?X[foo->moo,abc->cde(?Y)]),\naf w(?X).
```

Subsequent chapters describe many additional constructs that can appear in rule bodies. Also, compared to FLORA-2, ERGO supports much richer syntax in the rule *heads* as well, including disjunction and quantifiers.

As usual in logic languages, a single rule with a disjunction in the body

```
head :- John[age->31],
        (John[children->{Bob,Mary}] ; John[children->Bill]).
```
(1)

is equivalent to the following pair of rules:

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

Disjunction is also allowed inside frame literals. For instance, rule (1) can be equivalently rewritten as:

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

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

- *Knowledge bases and queries*: A *knowledge base* 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.

  Note: In the FLORA-2 shell, the "?-" query prefix should not be used, as the shell expects queries only anyway, so putting the prefix in the shell is redundant and will cause a syntax error.

Example 8.1 (Publications Database) Figure 1 depicts a fragment of a FLORA-2 knowledge base that represents a database of scientific publications.

Schema:
paper[|authors => person, title => string|].
journal_p :: paper[|in_vol => volume|].
conf_p :: paper[|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

8.2 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*, which are also called Prolog atoms. They are typically used as names of predicates and function symbols. All FLORA-2 symbols belong to the class `\symbol`.

FLORA-2 also recognizes escape sequences inside single quotes (`'`, symbols). An escape sequence begins with a backslash (\). Table 1 lists the special escape character sequences and their corresponding special symbols. An escape sequence can also represent a Unicode character. Such a character is preceded with a backslash followed by the letters `x` or `X` followed by 1 to 6 *hexadecimal* digits (0–F) representing the character's Unicode value; or by the letters `u` or `U` followed by 1-7

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

Table 1: Escaped Character Sequences and Their Corresponding Symbols

decimal digits. The sequence of digits **must** be terminated with a vertical bar, |. The number that follows `u` or `x` must not exceed 1,114,112—the largest Unicode codepoint. For example, `\xd|` and `\u13|` represent the ASCII character Carriage Return, `\x3A|` and `\u58|` represents the semicolon, while `\x05D0|` and `\u1488|` are Unicode for the Hebrew letter Alef. In other contexts, a backslash is recognized as itself.

If it is necessary to include a 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'`.

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.

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

Table 2: Representation of Integers

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

Floating point 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 39.

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 as whitespace by the compiler. Therefore, tokens can also be delimited by comments.

8.3 Operators

As in Prolog, FLORA-2 allows the user to define operators, to make the syntax more natural. There are three kinds of operators: infix, prefix, and postfix. An infix operator appears between its two arguments, while a prefix operator appears in front of its single argument. A postfix operator is written 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 than or equal* to 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 uses (*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 may conflict with those of FLORA-2 and, as a result, your specification 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.

8.4 Logical Expressions

In a FLORA-2, 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 made with the infix operator "`,`" and disjunction is made using the infix operator "`;`". Negation is specified using the prefix operators "`\+`" and "`\naf`".[8] 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, \naf d`, is equivalent to the logical formula: $(\texttt{a} \wedge \texttt{b}) \vee (\texttt{c} \wedge (\neg \texttt{d}))$.

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

```
o[\naf att1->val1, att2->val2; meth->res]
```

is equivalent to the following formula:

```
(\naf o[att1->val1], o[att2->val2]) ; o[meth->res]
```

8.5 Arithmetic (and related) 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

[8] In brief, "`\+`" represents Prolog-style negation, which does not have an acceptable logical semantics. It is useful, however, when applied to non-tabled Prolog predicates, F-logic frames, or HiLog predicates. "`\naf`", on the other hand, is negation that implements the logical well-founded semantics. Refer to Section 20 for more information on the difference between negation operators.

the value 7. In addition, ERGO provides a powerful feature of inline evaluation of such expressions, which allows these expressions to appear as arguments to predicates and frames, and be automatically evaluated at runtime. Ergo also provides a number of additional operators, including list/set append/union/intersect, and difference, and string concatenation, which also automatically converts arbitrary terms to their printable form.

When dealing with arithmetic expressions, the order of literals is *sometimes* important. The comparison and evaluation operators for which the order is unimportant (the *logical* operators) are:

- `>`, `<`, `=<`, `>=`
- `!=`, `!==`, `=:=`, `=\=`
- `\is`
- ∼, \∼, `!`∼ (the last two being synonymous).

The operators for which the order is important (the *non-logical* operators) are:

- `==`
- `\=`, `\==`, `?=`
- `@<`, `@>`, `@=<`, `@>=`.

Logical operators commute among themselves, but non-logical operators generally do not commute with either logical or non-logical operators, and different orders of these operators in an expression may produce different results. For instance, if `?X` is not bound then `?X` == `abc`, `?X` = `abc` will fail, while `?X` = `abc`, `?X` == `abc` will succeed with `?X` = `abc`. The reason for this is, of course the non-logical operator ==.

Arithmetic expression must be instantiated at the time of evaluation. Otherwise, a runtime error will occur. However, FLORA-2 tries to delay the evaluation of arithmetic expressions until the variables become bound and it will issue a runtime error only if it determines that some variable will never get bound. For instance,

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

will *not* produce an error, while the following query will:

```
?- ?X > 1.
```

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]`, where `?X` and `?Y` are new variables. More detailed discussion of path expressions appears in Section 9.

In arithmetic expressions, 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 the 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 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, does the expression

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

correspond to the arithmetic expression `(1.m)+(2.n.k)`, to the path expressions `(1.m+2.n).k`, by `(1.m + 2).n.k`, or to `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; used to change precedence	not applicable	not applicable
not applicable	`.`	decimal point	not applicable	not applicable
	`.`	object reference	left	binary
400	`: ::`	class membership and subclass relationships	left	binary
600	`-`	minus sign	right	unary
	`+`	plus sign	right	unary
700	`*`	multiplication	left	binary
700	`**`	power	left	binary
	`/`	division	left	binary
800	`- +`	subtraction and addition	left	binary
	`=<`	less than or equals to	not applicable	binary
	`>=`	greater than or equals to	not applicable	binary
1000	`=:=`	numeric equals-to	not applicable	binary
	`=\=`	unequal to	not applicable	binary
	`\is`	arithmetic assignment	not applicable	binary
	`=`	unification	left	binary
	`!= or \=`	disunification	left	binary
	`==`	identity	left	binary
	`=..`	meta decomposition	left	binary
	`!== or \==`	not identical	left	binary
	`@<`	lexicographical less-than	left	binary
	`?=`	identical or not unifiable	left	binary
1200	`\naf`	well-founded negation	not applicable	unary
	`\neg`	explicit negation	not applicable	unary
	`\+`	Prolog-style negation	not applicable	unary
1250	$\sim$	semantic unification	left	binary
	$! \sim$ or $\setminus \sim$	semantic disunification	left	binary

Table 3: Operators in increasing precedence order

`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 tokens.

In addition to the operators, the builtin function, and constants listed in Table 4 can occur in arithmetic expressions, i.e., on the right side of `\is/2` and on either side of the inequalities (>, <, =<, and >=).

Note: the expression on the right side of `\is/2` must not contain any variables at the time the

`\is/2` predicate is evaluated. Otherwise, an error is issued. If you want to solve equations and constraints (e.g., find $?X$ such that $5 = ?X + 2$), `\is/2` is a wrong predicate to do so: see Section 32 for the proper way to do this.

Function	Arity	Meaning
min, max	2	minimum and maximum of the arguments
abs	1	absolute value
ceiling, floor, round	1	ceiling, floor, and rounding of a real
float	1	convert to float
truncate	1	truncate the decimal part of a real
mod	2	integer division modulo
rem	2	remainder of integer division
div, `%%`	2	integer division
exp, **	2	exponent of argument 1
exp	1	exponent of number E
sqrt	1	square root
sign	1	sign of number (1 or -1)
sin, cos	1	sine and cosine
asin,acos	1	arcsine and arccosine
tan, atan	1	tangent and arctangent
log	1	logarithm base E
log10	1	logarithm base 10
$\wedge$, $\vee$	2	bit-wise AND and bit-wise OR
\e, \pi	0	the E and PI numbers

Table 4: Arithmetic functions that can be used in arithmetic expressions

Also, in the arithmetic expressions on the right side of `\is/2`, both `pi`, `e`, `\pi`, and `\e` are recognized. However, outside of the context of `\is/2`, only `\pi` and `\e` are recognized as special constants. The other functions in the above table (like `min`, `max`, trigonometry) are recognized both in `\is/2` expressions and inside constraints (see Section 32). Yet others (`div`, `rem`, `mod`) are recognized only inside `\is/2` expressions.

ERGO also supplies additional functions that can be used with `\is`, including `sum`, `min`, `max`, `avg`, `last`, `count`, `delete`, `reverse`, and `nth`.

8.6 Quasi-constants and Quasi-variables

In some cases the developer might require the knowledge base to refer to the information about the source code where the various FLORA-2 statements occur or some other such information. To this end, the compiler provides a number of *quasi-constants*, which get substituted for real constants at compile or a loading time. Since these symbols' actual value depends on the context, they are called *quasi*-variables and not variables. The supported constants are:

- `\#`, `\#1`, `\#2`, etc. – Skolem constants.

- There also are so-called "global" Skolem quasi-constants – see Section 14.
- `\@!` – the Id of the rule where this quasi-constant occurs.
- `\@` – the module into which the file containing this quasi-constant is loaded. In the FLORA-2 shell, this quasi-constant is set to be the symbol `main`.
- `\@F` – the file in which this quasi-constant occurs. In the FLORA-2 shell, this quasi-constant is set to be the symbol `'(runtime)'`.
- `\@L` – the line in the source code on which this quasi-constant occurs.
- `\@?` – the quasi-constant that represents the null value. The null value quasi-constant has the special property that `\@? = \@?` and `\@? :=: \@?`, but not `\@? == \@?`. It is convenient to use `\@?` as a default value for various properties inherited from classes. This quasi-constant is also used in various interfaces to external sources, such as the database interface, JSON, and tabular data (CSV, DSV, etc.).

Some of these constants are further illustrated in later in this manual. A typical use of these constants is to put them somewhere in place of a constant or a variable. For instance,

```
p(\@F,?X,?Y) :- ?X = \@L, ?Y = \@!.
```

In addition, FLORA-2 provides a number of *quasi-variables*. Quasi-variables are similar to quasi-constants in that it is FLORA-2 that instantiates them with values. However, this happens at *run time*, during the evaluation. The supported quasi-variables are:

- `\?C` – the module from which the given rule was called. Used in the bodies of rules and queries.
- `\?F` – the file name from which a sensor was called. Used only in the bodies of the rules that are part of a sensor definition.
- `\?L` – the line number from which a sensor was called. Used only in the bodies of the rules that are part of a sensor definition.

8.7 Synonyms

For better readability by non-experts and for documentation purposes, FLORA-2 provides the following useful mnemonic keywords, which can be used in place of the most commonly used symbols:

Symbol	Synonym
:	\isa, \memberof
::	\sub, \subclassof
=>	\hastype
->	\hasvalue
+>>	\contains

8.8 Reserved Symbols

$\mathcal{F}$LORA-*2* reserves all symbols that begin with the backslash, such as `\or`, `\hastype`, `\io`, `\neg`, etc. In addition, the symbols `->`, `=>`, `==>`, `<==`, `<==>`, `~~>`, `<~~`, `<~~>`, `:-`, `-:`, `!-`, `?-`, `?`, `|`, `#`, ",", ";", ":", "::", `->->`, `-»`, `+»`, arithmetic operators, `\/`, and `/\` are also reserved and cannot be used as names of methods, function symbols, predicates, and the like.

9 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 [7]. The basic idea is to allow the following *path expressions* wherever Id-terms are allowed:

7. `O.M`

The path expression in (7) refers to an object $\mathtt{R_0}$ for which $\mathtt{O[M->R_0]}$ holds. The symbols `O` and `M` stand for an Id-term or a path expression. As a special case, `M` can be a method that takes arguments. For instance, $\mathtt{O.M(P_1,\ldots,P_k)}$ is a valid path expression.

Path expressions associate to the left, so `a.b.c` is equivalent to `(a.b).c`, which specifies the object `o` such that $\mathtt{a[b->x]} \wedge \mathtt{x[c->o]}$ holds (note that $\mathtt{x} = \mathtt{a.b}$). To change that, parentheses can be used. For instance, `a.(b.c)` is that object `o1` for which $\mathtt{b[c->x1]} \wedge \mathtt{a[x1->o1]}$ holds (note that in this case, $\mathtt{x1} = \mathtt{b.c}$). 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: Since a path expression represents an object Id, it can appear wherever an oid can, and it *cannot* appear in place of a truth-valued expression (e.g., a subquery). Thus,

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

is illegal. 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 frames can be arbitrarily nested, this leads to a concise and flexible specification language for object properties, as illustrated in the following example. □

Example 9.1 (Path Expressions) Consider again the schema given in Figure 1. If $?n$ represents the name of a person, the following path expression is a query that returns all editors of conferences in which $?n$ had a paper:

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

Likewise, the answer to the query

```
?- ?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:

```
?- ?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.[9]

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

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

and then simply write ...`[self->?O]`... anywhere in a complex path expression. This would bind the Id of the current object to the variable `?O`.

[9] A similar feature is used in other languages, e.g., XSQL [9].

Example 9.2 (Path Expressions with self) To illustrate the convenience afforded by the use of the `self` attribute in path expressions, consider the second query in Example 9.1. 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].
```

10 Set Notation

The original F-logic [10] permitted convenient set notation as return values of set-valued methods. For instance,

```
John[children->{Mary,Bob}]
```

is a shortcut for the conjunction

```
John[children->Mary], John[children->Bob]
```

whether this expression occurs in the head or the body of a rule. *FLORA-2* makes a leap forward in this direction and permits set notation anywhere a path expression is allowed. Here are some examples of what is possible:

```
{Mary,Joe}:{Student,Worker}.
happily_married(?X,?Y) :- person({?X,?Y}), ?X[{spouse,loves}->?Y].
child(John,{Mary,Kate,Bob}).
```

The above statements are, respectively, shortcuts for

```
Mary:Student, Mary:Worker, Joe:Student, Joe:Worker.
happily_married(?X,?Y) :- person(?X), person(?Y),
                          ?X[spouse->?Y], ?X[loves->?Y].
child(John,Mary), child(John,Kate), child(John,Bob).
```

For some more extreme examples, consider these:

```
{p,q}(a,g(f({b,c}))).
r({?X,?Y}) :- {p,q}(?,?(f({?X,?Y}))).
{a,m}[{prop1,prop2} -> {1,{2,3},4}].
```

These are shortcuts for the following statements, respectively:

```
p(a,g(f(b))), p(a,g(f(c))), q(a,g(f(b))), q(a,g(f(c))).
r(?X), r(?Y) :- p(?,?(f(?X))), p(?,?(f(?Y))), q(?,?(f(?X))), q(?,?(f(?Y))).
a[prop1->{1,2,3,4}, prop2->{1,2,3,4}], m[prop1->{1,2,3,4}, prop2->{1,2,3,4}].
```

The set term `{1,{2,3},4}` above is just a less readable way of writing `{1,2,3,4}`, by the way—there are no true set-objects in $\mathcal{F}$LORA-*2* or F-logic. Other extremely useful cases of the use of the set notation involve equality and related operators. For instance,

```
?X={?Y,?Z,?W}
?X!={1,2,3}
```

instead of the much more tedious

```
?X=?Y, ?Y=?Z, ?Z=?W
?X!=1, ?X!=2, ?X!=3
```

11 Typed Variables

Apart from the regular variables, $\mathcal{F}$LORA-*2* supports *typed variables*. Typed variables can be bound to classes, which will restrict them so that they will be unifiable only with members of the specified classes. This includes the classes associated with all the primitive data types such as `\integer`, `\real`, `\time`, `\list`, `\duration`, etc., which are discussed in Section 39, and user-defined classes. The syntax is `?Varˆˆ Class`. The class can be a class name (which may have variables) or a class expression. For instance,

```
father(?Xˆˆ\integer,?Yˆˆ Person) :- parent(?X,?Y), male(?X).
grandfather(?Xˆˆ\integer,?Yˆˆ Person) :- father(?X,?Zˆˆ Person),parent(Z,Y).
?- foo(?Yˆˆ(A,(B-C) ; \dateTyme).
```

As seen from the example, the type declaration can appear both in the head and the body of a rule. The semantics is that once the variable is bound then the binding is checked for belonging to the specified class. Theoretically `?Xˆˆ Cl` is equivalent to making a test like `?X:Cl` *once* `?X` *gets bound by a subgoal*, but in practice typed variables can be much more efficient. Testing `?X:Cl`, where `?X` is unbound, involves enumerating the entire class `Cl`, which can be expensive when the class is large and may not terminate, if the class is infinite (e.g., `?X:\integer`). In contrast, `?Xˆˆ Cl` does not involve any tests until `?X` gets bound. This is convenient because often the knowledge engineer

does not know which subgoal will bind `?X` and so the guessing game of where to put the test `?X:Cl` is avoided. However, the knowledge engineer must be certain that `?X` *will* get bound at some point or else no checks will be performed whatsoever (because `?X^^Cl` causes the membership test for `Cl` to be performed only when `?X` gets bound to a non-variable).

The expression `?Varˆˆ Class` should be understood as a declaration, so it should be done only once per variable in the same rule. Multiple declarations are treated as intersections. For instance,

```
?- ?X^^foo=?, ?X^^moo=?.

?X = ?_h6272 { \$typed variable : type = (moo  ',’  foo) }

?- ?X^^foo=?Y^^moo.

?X = ?_h5593 { \$typed variable : type = (moo  ',’  foo) }
?Y = ?_h5593 { \$typed variable : type = (moo  ',’  foo) }
```

Here are some additional examples:

```
{1,2}:foo, {2,3}:moo.                          // facts

?- ?^^foo = 4.                                 // false
?- ?^^foo = 1                                  // true
?- ?X^^foo = ?^^moo, ?X=3                      // false
?- ?X^^foo = ?^^moo, ?X=2                      // true
?- ?^^(foo-moo) = 2  // false
?- ?^^(foo-moo) = 1  // true
?- ?^^((foo-moo);(moo-foo)) = 2                // false
?- ?^^((foo-moo);(moo-foo)) = 3                // true
?- ?X^^\real=?, ?X^^\short=1                   // false
?- {1,2,-3}:{\integer,\short,\long},           // true
      {1.2,3.4}:{\real,\float,\double},
      [a,3] = ?X^^\list,
      ?X:\list,
      "abc"^^\charlist=?Y,
      ?Y:\charlist,
      writeln(test6=ok)@\plg.
```

Quantified typed variables, i.e., typed variables in the lists attached to quantifiers are also allowed. For instance, the query

```
pp({a,b}). qq(a).
?- forall(?X)^(pp(?X) ~~> qq(?X)).
```

is false because for `?X=b` the proposition `qq(b)` is false. However, the query

```
pp({a,b}). qq(a). a:foo.
?- forall(?X^^foo)^(pp(?X) ~~> qq(?X)).
```

is true because for all `?X`'s that are in class `foo` and for which `pp(...)` is true, `qq(...)` is also true. The above can be seen as just a more convenient form than

```
?- forall(?X)^(pp(?X^^foo) ~~> qq(?X)).
```

but otherwise they are equivalent.

12 Truth Values and Object Values

Id-terms, terms and path expressions can be also understood as objects. This is clear for Id-terms. The object interpretation for path expressions of the form (7) was given on page 32. On the other hand, frame formulas, class membership, and subclassing are typically understood as truth-valued formulas. However, there also is a natural way to interpret them as objects. 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 frames. Consequently, all frame literals of the form (1) through (7) have dual readings: 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 knowledge base, 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 of the language.

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

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

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

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:

$$\mathit{obj}(\texttt{d.e}) = \{\texttt{x}_{\texttt{de}} \mid \mathcal{I} \models \texttt{d[e -> x}_{\texttt{de}}\texttt{]}\}$$

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 frame literals to sets of ground oids (t, o, c, d, m can be oids or path expressions):

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

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

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 its 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).
```

Here `$a[b->?X]` is a `reification` of the formula `a[b->?X]` (see Section 19.2), i.e., an object that represents the formula. FLORA-2 does not allow one to write `foobar(a[b->?X],?X)` because this notation proved to be error-prone and confusing to the user in the previous versions of the system.

13 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)].
```

However, `John[is_tall->void]` is *not the same* as `John[is_tall]`: one must use one form or the other consistently. Conceptually, `John[is_tall->void]` defines a property that returns the value `void`, but could, in principle, return other values as well. In contrast, in `John[is_tall]`, the property `is_tall` does not return anything. So, `John[is_tall]` does not entail `John[is_tall->void]` and vice versa.

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 inherited, if specified as part of the class information:

```
Buddhist[|is_vegetarian|].
John:Buddhist.
```

The above says that all Buddhists are vegetarian by default and John (the object with oid `John`) is a Buddhist. Since `is_vegetarian` is specified as a property of the entire class `Buddhist`, it follows that John is also a vegetarian, i.e., `John[is_vegetarian]`.

13.1 Boolean Signatures

Boolean methods can have signatures like the value-returning methods. These signatures can be specified as part of the object-level information (in which case they apply to specific objects) or as part of the class information (in which case they apply to all objects in the class and to all subclasses):

```
Obj[=>Meth]
Class[|=>Meth|]
```

The first statement refers to `Class` as an individual object, while the second is a statement about the type of the object `Class` as a class. It thus is inherited by every member of `Class` and all its subclasses. For instance,

```
Person[|=>loves(game)|].
```

14 Skolem Symbols

For applications where the particular names for oids are not important, FLORA-*2* provides the quasi-constants `\#`, `\#1`, `\#2`, `\#abc`, etc., or `\##1`, `\##2`, `\##abc`, etc. (note: no `\##`!), to automatically generate a new *Skolem constant* or a *Skolem function symbol* to test for such automatically generated constants. We call such symbols *Skolem symbols*. Skolem symbols are interpreted *differently* in the rule heads and bodies.

Skolem symbols can be *local*—those that are prefixed with `\#`—and *global*—the ones prefixed with `\##`. We first describe local Skolems.

The quasi-constant `\#` is a special *unique local Skolem*, `\#1`, `\#2`, etc., are *numbered local Skolems*, and `\#abc` is a *named local Skolem*. For global Skolems, FLORA-*2* has *numbered global Skolems* and *named global Skolems* only.

Local unique Skolems in rule heads and descriptors outside of the reification operator. Outside of reification (see Section 19.2 for the details of reification), the head-occurrences of Skolem symbols are interpreted, as described below.

Each occurrence of `\#` in the rule head or a rule descriptor (see Section 36 to learn about rule descriptors) represents a *new* Skolem constant, which is unique throughout the source code.

Uniqueness is achieved through the use of a 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 example

```
\#[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.

Note: a unique Skolem quasi-constant cannot match any other constant or quasi-constant anywhere in the knowledge base—not even in the same clause.

Co-reference of local Skolem symbols: numbered and named Skolems. In some situations, it is necessary to be able to create a new oid and use it within the same rule head or a fact multiple times. Sometimes the *same* Skolem might need to be used across *several* different rules and facts. Since such an oid needs to be referenced inside the same clause, it is no longer possible to use `\#`, because each occurrence of `\#` causes the compiler to generate a new Skolem symbol. To solve this problem, $\mathcal{F}$LORA-*2* allows *numbered Skolem symbols* as well as *named Skolem symbols*, which have the form `\#132` or `\#abc`, *i.e.*, `\#` with a number or an alphanumeric symbol attached to it. For instance,

```
\#1[ssn->123,father->f(\#1)[name->John,spouse->\#[name->Mary,knows->\#foobar2]]].
\#foobar2[self->\#foobar2,child->\#1].
```

The first time the compiler finds `\#1` in the first clause above, it will generate a new Skolem constant for an oid. However, the second occurrence of `\#1` in the *same* clause (in `f(\#1)`) will use the oid previously generated for the first occurrence. Similarly, the two occurrences of `\#foobar2` in the second clause refer to the same new constant. On the other hand, occurrences of `\#1` and `\#foobar2` in *different* clauses stand for different oids. Thus, the occurrences of `\#1` (and of `\#foobar2`) in the first and second clauses above refer to different objects.

The same numbered or named Skolem can be co-referenced in *different* facts and rules. The general rule is that the scope of a numbered Skolem is a $\mathcal{F}$LORA-*2* sentence terminated with a period. For instance, in the following

```
(h1(?X,\#1) :- b1(?X),c1(?X)),
p(a,\#1),
q(b(\#1)),
(h2(?X,\#1,?Y) :- b2(?X,?Y)).
```

the four occurrences of `\#1` represent the same Skolem constant. Note that the first and the last statement above are rules, while the two middle statements are facts. The reason why all occurrences of `\#1` are the same is because all four statements are withing the same scope: all four are terminated by the same period. Note also that each rule must be enclosed in parentheses or else ambiguity arises. For instance,

```
h1(?X,\#1) :- b1(?X),c1(?X),
```

```
p(a,\#1),
q(b(\#1)).
```

would be interpreted as a single rule whose body has four literals, *not* as a statement that contains a rule `h1(?X,\#1) :- b1(?X),c1(?X)` and two additional, separate facts. Also note that it is not necessary to put each of the four sub-statement above on a separate line—this was done for readability.

Note: although using local numbered/named Skolems across different rules, as we have just shown, is possible, it is generally inconvenient and error-prone. We recommend global Skolems for that—see next.

User-controlled scope of co-reference of Skolem symbols: global Skolems. When the scope of a Skolem quasi-constant must cover several rules or facts, the most convenient way is to use *global Skolems*. *Global Skolems* are always numbered or named (no `\##`), and the scope of the these symbols is controlled by the programmer explicitly.

Global Skolems have the form `\##Number` or `\##Name`, where *Number* is an integer and *Name* is an alphanumerical symbol. For instance, in our earlier example,

```
\##1[ssn->123,father->f(\##1)[name->John,spouse->\#[name->Mary,knows->\##foobar2]]].
\##foobar2[self->\##foobar2,child->\##1].
```

Here *all* occurrences of `\##1` refer to the same unique constant and all occurrences of `\##foobar2` refer to the same unique constant (but different from the one for `\##1`).

The scope of co-reference for global Skolems persists until it is changed explicitly with the compile-time directive

```
:- new_global_oid_scope.
```

To use this effectively, one must understand when these directives take effect. The following example may help clarify this. Suppose we load a file with the following contents

```
p(\##abc).                  // statement 1
:- new_global_oid_scope.    // statement 2
p(\##abc).                  // statement 3
?- insert{p(\##abc)}.       // statement 4
:- new_global_oid_scope.    // statement 5
?- insert{p(\##abc)}.       // statement 6
p(\##abc).                  // statement 7
```

```
p(\##abc).                    // statement 8
?- insert{p(\##abc)}.         // statement 9
```

If we now ask the query

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

we will get these three distinct answers (up to a renaming of the Skolem constants):

```
?X = \##abc  (_$_$_ergo'autogen1|abc'2)
?X = \##abc  (_$_$_ergo'autogen2|abc'2)
?X = \##abc  (_$_$_ergo'autogen3|abc'2)
```

Why only three answers? At compile time, `\##abc` in statements 1, 3, 6 will be replaced with three different Skolems because of the two intervening compile time directives `new_global_oid_scope`. Statements 4 and 7–9 will generate no new oids and therefore will produce duplicate facts, which will be discarded. (The occurrence of `\##abc` in statement 4 will be replaced with the same constant as in statement 3 and the occurrences in statements 7–9 will be replaced with the same constant as in statement 6).

In some very rare cases, one might need to employ the runtime version of the `new_global_oid_scope` directive:

```
?- new_global_oid_scope.
```

This may be needed, for instance, to ensure that two different files share no global Skolems. Such an effect can be achieved by executing the above runtime directive before compiling the second file.

Generating Skolems at run time. Normally, Skolem constants are generated at compile time without regard for the oids that might exist at run time. Sometimes it is necessary to generate a Skolem constant at *run time*. This can be accomplished with the `skolem{...}` built-in. For instance,

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

?X = _$_$_flora'dyn_skolem308   (#)

1 solution(s) in 0.0000 seconds

Yes
```

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

?X = _$_$_flora'dyn_skolem309  (#)
```

etc.

Local Skolems in subgoals of rule bodies outside of the reification operator. Local Skolem quasi-constants that appear in the *body* of a rule or in a query are interpreted *differently*. This is because, if local Skolems in rule bodies were interpreted as unique new constants, such a constant would never match anything else in a knowledge base and thus the rule will never "fire" and the query will return no results. (Global Skolems are different in this respect, since they *can* patch constants that appear elsewhere in the knowledge base.)

In a body-subgoal outside of the reification operator, a Skolem symbol (numbered/named or not) is interpreted differently that in the head: not as a new Skolem constant, but as a *test* of whether or not the corresponding argument is bound to a Skolem constant. One can think of it as a *variable* that is tested for being bound to a Skolem constant. Numbered Skolems within the same rule are interpreted as the same variable, so the occurrences of the same numbered or named Skolem are expected to be bound to the same constant. For instance, in

```
?- insert{abc[prop1->\#, prop2->\#3, prop3->\#3],
          cde[prop1->\#, prop2->\#3, prop3->\#]}.
test1 :- abc[prop1->\#, prop2->\#abc, prop3->\#abc].
test2 :- cde[prop1->\#, prop2->\#5, prop3->\#5].
```

Here the query `test1` will return `true` because in the object `abc` the properties `prop1` and `prop2` are bound to the same Skolem constant. In contrast, in `cde`, these properties are *not* bound to the same Skolem (since `\#3` and `\#` are different Skolems), so the query `test2` fails.

Note that, as a test, both `\#` and `\#`N, where N is an positive integer, match not only Skolem constants, but also Skolem functions. For instance, the query

```
?- insert{p(\#(a,b))}, p(\#).
```

succeeds because `\#(a,b)` is a term obtained from an application of a Skolem function, `\#`, to its arguments.

Note that, since Skolems have a different semantics in the rule head and in the body, the following, somewhat counter-intuitive situation might occur:

```
p(\#1) :- q(\#1).
```

Here `\#1` inside `p(...)` is a constant, while in inside `q(...)` it is a variable with a restriction that is can be bound only to a Skolem term.

Local Skolems in reified rules. If a reified rule, R, occurs somewhere in the body or a head of another, normal rule (or fact), the local Skolem symbols that occur in R are interpreted the same way as if the rule were outside of reification. That is, the head occurrences in R are interpreted as Skolem constants and the body occurrences are interpreted as variables that can be bound only to Skolem terms. For instance, in

```
head1(${head11(?X,\#) :- body11(\#1,\#1)}).
head2 :- body2(${head22(?X,\#) :- body22(\#1,\#1)}).
```

the symbol `\#` in the fact `head11(...)` is a Skolem constant, while the two occurrences of the symbol `\#1` in `body11` test if `body11(?X,?Y)` can succeed with `?X` and `?Y` bound to the same Skolem constant. The interpretation of `\#` and `\#1` in the second rule above is similar.

Local Skolems in reified subgoals. If a reified subgoal (not rule), G, contains a local Skolem symbol, that symbol is interpreted as a Skolem constant. This is true both if G occurs in the head of a rule and in the body. Note that this implies that if one later uses G as a *query* then this query is likely to fail, as a local Skolem is unlikely to match anything else in the knowledge base. For instance, in

```
?- insert{p(a), p(\#)}.
?- ?X = ${p(\#)}, ?X.
```

the second query will fail because the occurrence of `\#` there is a Skolem constant that differs from the Skolem constant in the first rule. However,

```
?- ?X = ${p(\#)}, insert{?X}, p(\#).
```

succeeds because when the reified subgoal `${p(\#)}` is inserted into the database, the local Skolem symbol is interpreted as a constant and the subsequent check `p(\#)` tests that the argument of `p` is bound to a Skolem constant.

Numbered/named Skolems are interpreted the same way as non-numbered/unnamed ones except that different occurrences of the same numbered/named Skolem symbol are treated as identical within the same rule or query. For instance:

```
?- ?X = ${p(\#2,\#2)}, insert{?X}, p(\#7,\#7).
?- erase{p(?,?)}, ?X = ${p(\#2,\#)}, insert{?X}, p(\#7,\#7).
```

Here the first query succeeds, while the second fails.

Global Skolems in rule bodies and queries. Unlike local Skolems, global ones have the same meaning in queries, rule bodies, rule heads, and facts. However, one must always keep the scope of these quasi-constants in mind because the same global Skolem, say `\##abc`, before and after the directive `new_global_oid_scope` denotes different constants. So, for example, in

```
p(\##abc).
?- p(\##abc).
:- new_global_oid_scope.
?- p(\##abc).
```

the first query will return "Yes" but the second "No."

More on Skolems. A Skolem constant—both local and global—can also be used as a function symbol and even a predicate symbol. For instance, `\#(a,b)`. Since *F*LORA-*2* terms are actually HiLog terms, we can also have higher-order Skolem functions:

```
\#1(foo,bar)(123)
\#(a,c,d)(o,w)(1,2)
```

Sometimes it is useful to know whether a particular term is a *Skolem term*, i.e., whether it is a Skolem constant or is constructed using a Skolem function or a higher-order Skolem function. To this end, *F*LORA-*2* provides a special built-in class, `\skolem`. For instance, in the following case

```
p(\#1(foo,bar)(123)).
?- p(?X), ?X:\skolem.
```

the variable `?X` gets bound to a higher-order Skolem term, so the query succeeds. But the query `?- f(a):\skolem` fails, since `f` is not a Skolem symbol. In addition to this, *F*LORA-*2* provides a builtin for checking if a particular symbol (symbol, not just any term) is Skolem. For instance,

```
?- skolem{?X}, isskolem{?X}.
```

will succeed. Here `skolem{...}` generates a new Skolem constant and `isskolem{...}` tests if `?X` is bound to a Skolem constant. Of course, `?- skolem?X, ?X:\skolem` will also succeed, but the

`\skolem` class contains all Skolem terms, while the `isskolem{...}` primitive is true of Skolem constants only.

Finally, we note that both the `isskolem{...}` primitive and the `\skolem` class require that the argument (the class instance being tested) must be bound.

15 Testing Meta-properties of Symbols and Variables

Sometimes it useful to be able to be able to find out to what kind symbol a particular variable is bound or even whether that variable is bound to a term or not. In Section 14, we already saw the `isskolem{...}` primitive, which can tell whether a particular symbol is a Skolem constant. Here is a summary of all such meta-predicates. For the meaning of the IRI and string constants, please refer to Section 39.

- `isnumber{Arg}` — tests whether the argument is (or is bound to) a number.
- `isinteger{Arg}` — tests whether the argument is (or is bound to) an integer number.
- `isfloat{Arg}` — tests whether the argument is (or is bound to) a floating point number.
- `isdecimal{Arg}` — tests if the argument is (or is bound to) a decimal number. At present, this is the same as `isnumber{Arg}`.
- `isatom{Arg}` — tests whether the argument is (or is bound to) a Prolog atom.
- `iscompound{Arg}` — tests whether the argument is (or is bound to) a compound term, i.e., a term that has a non-zero-ary function symbol.
- `isatomic{Arg}` — tests if the argument is (or is bound to) a Prolog atom or a number.
- `islist{Arg}` — tests if the argument is (or is bound to) a list term.
- `ischarlist{Arg}` — tests whether the argument is (or is bound to) a list or ASCII characters.

- `isiri{Arg}` — tests whether the argument is (or is bound to) an IRI data type.
- `isstring{Arg}` — tests whether the argument is (or is bound to) a string data type.
- `issymbol{Arg}` — tests whether the argument is (or is bound to) an abstract symbol. An abstract symbol is any atom that is not an internal representation of a string or an IRI. (These internal representations involve special unprintable characters and thus are unlikely to be used by a normal user directly.)

- `isvar{Arg}` — tests if the argument is an unbound variable.
- `isnonvar{Arg}` — tests if the argument is not an unbound variable.
- `isground{Arg}` — tests if the argument is a ground term (or is bound to one).
- `isnonground{Arg}` — tests if the argument is not a ground term.
- `isskolem{Arg}` — tests if `Arg` is bound to a Skolem constant.
- `variables{Term,List}` — binds `List` to the list of all the variables that occur in `Term`.
- `cloneterm{Term,ClonedTerm}` — creates a copy of `Term` with the same constants and function symbols, but variables are consistently renamed to become new variables.

In addition, some of the above primitives have *delayable* 2-argument versions. A delayable version differs in that if the first argument is a variable then evaluation of such a builtin is delayed until the argument gets bound. If the argument does not get bound at the end of the query computation, then the outcome depends on the second *mode*-argument: if the mode is `must` then an error is issued; if the mode is `wish` then the builtin is quietly evaluated to *false*. For instance,

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

No

flora2 ?-  isinteger{?X}, ?X=1.

No

flora2 ?-  isinteger{?X,wish}, ?X=1.

?X = 1

Yes

flora2 ?-  isinteger{?X,must}, ?X=1.

?X = 1

Yes

flora2 ?-  isinteger{?X,must}.
```

```
++Abort[Flora-2]> in file (runtime) on line 1: instantiation error in builtin:
                  isinteger{_h3464}; unbound argument

flora2 ?-  isinteger{?X,wish}.

No
```

Thus, the delayable versions of the above primitives are insensitive to the order in which they appear in the rule body, which makes them sometimes easier to use and renders their behavior more logical. On the other hand, the delayable versions cannot serve as *guards* for subsequent evaluations. For example, if `foo(?X)` expects `?X` to be an integer then

```
... :- ..., isinteger{?X,must}, foo(?X), ...
```

will not prevent calling `foo/1` with `?X` an unbound variable, since `isinteger{Arg,Mode}` will be delayed past the moment `foo(?X)` is evaluated. Using `isinteger{?X}` instead will do the job.

The delayable versions of the aforesaid builtins are listed below. The delayable versions of `isvar{Arg,Mode}`, `isnonvar{Arg,Mode}`, `isground{Arg,Mode}`, and `isnonground{Arg,Mode}` have slightly different semantics.

- `isnumber{Arg,Mode}` — the delayable version of `isnumber{Arg}`.
- `isinteger{Arg,Mode}` — the delayable version of `isinteger{Arg}`.
- `isfloat{Arg,Mode}` — the delayable version of `isfloat{Arg}`.
- `isdecimal{Arg,Mode}` — the delayable version of `isdecimal{Arg}`.
- `isatom{Arg,Mode}` — the delayable version of `isatom{Arg}`.
- `iscompound{Arg,Mode}` — the delayable version of `iscompound{Arg}`.
- `isatomic{Arg,Mode}` — the delayable version of `isatomic{Arg}`.
- `islist{Arg,Mode}` — the delayable version of `islist{Arg}`.
- `ischarlist{Arg,Mode}` — the delayable version of `ischarlist{Arg}`.
- `isiri{Arg,Mode}` — the delayable version of `isiri{Arg}`.
- `isstring{Arg,Mode}` — the delayable version of `isstring{Arg}`.

- `issymbol{Arg,Mode}` — the delayable version of `issymbol{Arg}`.
- `isvar{Arg,Mode}` — the delayable version of `isvar{Arg}`. If the argument `?X` in `isvar{?X,must}` *at the end of the query computation* is *not* an unbound variable, an error is issued. Otherwise, the result is true. For `isvar{?X,wish}`, the evaluation is delayed and then `?X` is tested. If it is unbound, the result is true; otherwise, it is false. No errors are issued.
- `isnonvar{Arg,Mode}` — the delayable version of `isnonvar{Arg}`. If `?X` in `isnonvar{?X,must}` *at the end of the query computation* is a variable, an error is issued. Otherwise, the result is true. For `isnonvar{?X,wish}`, the evaluation is delayed and then `?X` is tested. If it is bound, the result is true; otherwise, it is false. No errors are issued.
- `isground{Arg,Mode}` — the delayable version of `isground{Arg}`. If `?X` in `isground{?X,must}` *at the end of the query computation* is non-ground, an error is issued. Otherwise, the result is true. For `isground{?X,wish}`, the evaluation is delayed and then `?X` is tested. If it is bound to a ground term, the result is true; otherwise, it is false. No errors are issued.
- `isnonground{Arg,Mode}` — the delayable version of `isnonground{Arg}`. If `?X` in `isnonground{?X,must}` *at the end of the query computation* is ground, an error is issued. Otherwise, the result is true. For `isnonground{?X,wish}`, the evaluation is delayed and then `?X` is tested. If it is bound to a ground term, the result is false; otherwise, it is true. No errors are issued.

16 Lists, Sets, Ranges. The Operators \in, \subset, \sublist

List is a very important data structure in $\mathcal{F}$LORA-*2*, which has several useful representations as a term. The simplest is just an enumeration like `[a21,12,?X,cde,?Y,pqr]`, which contains five items: three symbols (`a21`, `cde`, `pqr`) one integer (12) and two variables (`?X`, `?Y`).

In addition, the term `[elt1,elt2,...,eltk | rest]` represents a list that has `elt1`, `elt2`, ..., `eltk` in the *front* (the `...` is not part of the syntax, but is intended to convey the fact that the front of the list can have arbitrary length) and `rest` is the tail of the list. For instance, the aforementioned list `[a21,12,?X,cde,?Y,pqr]` can be alternatively represented as `[a21,12,?X | [cde,?Y,pqr]]`, or `[a21,12 | [?X,cde,?Y,pqr]]`, or `[a21 | [12,?X,cde,?Y,pqr]]`, and in two more ways. Very often the term to the right of | is a variable, which is a convenient way to split off a tail of a list at a desired point. For instance,

```
..., [a21,12,33,cde,?Y,pqr] = [a21,12,?X|?Tail], ...
```

will unify the two list-terms and will result in `?Tail` getting bound to `[cde,?Y,pqr]` and `?X` to 33.

Set (along with maps, dictionaries) is a data structure that is supported in $\mathcal{ERGO}$ only. It provides for efficient ways to store and search unordered collections of data.

Range is a data structure that represents an ordered collection of numbers in a certain range or of characters in a range (usually ASCII characters, but more generally could be characters in any supported character set. Examples include `23..76` (integers between 23 and 77), `2.34..7` (numbers 2.34, 3.34, 4.34, 5.34, 6.34), `7..5` (numbers can go down: 7, 6, 5), `b..d` (characters `b`, `c`, `d`), or `f..'?'` (characters `f`, `e`, `d`, ..., down to `'?'` in the ASCII table).

Note that the range facility is more flexible than it might look at first sight. In particular, one can use arithmetic expressions to generate ranges with step increments and more. For instance, if one wants some variable, `?Y` to range over the interval `10..16` with step increment of 2, this can be done as follows:

```
?- ?_X \in 0..3, ?Y \is 10 + ?_X*2, ...use ?Y here...
```

and `?Y` will be successively bound to 10, 12, 14, 16.

The \in operator. This operator allows one to test whether a term unifies with a member of a list, of a range, or (in $\mathcal{ERGO}$) of a set:

```
?- 5 \in 2..9, abc \in [2,3,abc,9].  // is true
?- f(?X) \in set123.  // ?X = 3, if set123 represents an Ergo set that has f(3)
```

Another important use of the `\in` operator is to successively bind a variable to every member of a list, range, or a set. For instance,

```
?- ?X \in a..k, p(?X),
   ?Y \in [abc,cde,efg], q(?Y).
```

successively binds `?X` to the symbols `a`, `b`, ..., `k` and checks if any of them are in predicate (or database table) `p` and then binds `?Y` successively to `abc`, `cde`, `efg` and checks if any of them are in `q`. The answer to this query is a set of binding pairs `?X=x`, `?Y=y` such that `x` is in the interval `a..k` and `p(x)` is true and `y` is one of the constants `abc`, `cde`, or `efg` such that `q(y)` is true.

The \subset operator. This operator is a convenient way to test if one list is a subset (ignoring the order of the elements) of another list. In $\mathcal{ERGO}$, it can also be used to test if one set is a subset of another. For instance,

```
?- [a,b(?X)] \subset [1,b(8),3,c,a].  // is true and ?X gets bound to 8
?- set1 \subset set2.  // true, if set1, set2 are Ergo sets and every
                       // member of set1 unifies with some member of set2
```

The \sublist operator. This operator applies to lists only. A subgoal like `list1 \sublist list2` checks if `list1` unifies with a sublist of `list2`. Checking for sublists preserves the order of the elements, but not adjacency. For instance,

```
?- [1,3,6,5] \sublist [1,2,3,4,6,7,5]. // true: same order, but not adjacency
?- [1,3,6,5] \sublist [1,2,3,4,5,6].   // false: order not preserved
```

17 Multifile Knowledge Bases

*F*LORA-*2* supports many ways in which a knowledge base can be modularized. First, it 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 file can be composed 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, *F*LORA-*2* knowledge bases can invoke Prolog modules and vice versa. In this way, a large system can be built partly in Prolog and partly in *F*LORA-*2*.

We discuss each of these modularization methods in turn.

17.1 *F*LORA-*2* Modules

A *F*LORA-*2* *module* is an abstraction that allows a large knowledge base to be split into separate libraries that can be reused in multiple ways in the same system. Formally, a module is a pair that consists of a *name* and a *contents*. The name must be an alphanumeric symbol (the underscore character, _, is also allowed), and the contents consists of part of the knowledge base that is typically loaded from some file (but can also be constructed runtime by inserting facts into another module).

The basic idea behind *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 knowledge base can then invoke this library's methods by providing the name of the module (and the method/predicate names, of course). The exported methods and predicates can be called by other parts of the knowledge base. (A module can have non-public methods, if the module is encapsulated — see Section 17.13.) 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 or added to any module and the same file can even be loaded/added to two different modules at the same time. The same module can be reused during the same *F*LORA-*2* session 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 FLORA-2, modules are completely decoupled from file names. A knowledge base knows only the module names it needs to call, but not the file names. Specific files can be loaded into modules by some other, unrelated bootstrapping code. Moreover, a knowledge base can be written in such a way that it invokes 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 FLORA-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.

FLORA-2 has *three kinds of modules*. 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 second kind of a module is a *Prolog module*. This is an abstraction in FLORA-2, which is used to invoke Prolog predicates. Prolog modules are static and are assumed to be closely associated with their code. We describe these modules in Section 17.8. (Do not confuse FLORA-2 Prolog modules — an abstraction used in the language of FLORA-2— with Prolog modules that are Prolog itself.) The third type of modules are the FLORA-2 *system modules*. These modules are preloaded with FLORA-2 and provide useful methods and predicates (*e.g.*, I/O) and, thus, are also static. These modules are described in Section 17.10 and 46. The abstraction of system modules is a convenience provided by FLORA-2, which enables users 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.

17.2 Calling Methods and Predicates Defined in User Modules

If *literal* is a frame 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.[10] 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,

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

```
..., ?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 knowledge bases do not know where they will be loaded, using the `@`*module* idiom to call a literal in the same module is difficult. However, it is easy to do with the help of the quasi-constant `\@`, 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 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.

2. Module specification is distributive over logical connectives, including the conjunction operator, "`,`", the disjunction, "`;`", and the negation operators, "`\+`" and "`\naf`". For example, the formula below:

   ```
   (John[father->Smith], \naf Smith[spouse->Mary]) @ foomod
   ```

   is equivalent to the following formula:

   ```
   John[father->Smith] @ foomod, \naf (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 frame 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

```
?- 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 19.2), as in

```
?- 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,

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

is equivalent to

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

17.3 Finding the Current Module Name

Since a $\mathcal{F}$LORA-*2* knowledge base can be loaded into any module, it does not have a priori knowledge of the module it will be executing in. However, the knowledge base can find its module at runtime using the quasi-constant `\@`, 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
```

17.4 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. Alternatively, the `\?C` quasi-variable can be used. For instance,

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

17.5 Loading Files into User Modules

*F*LORA-*2* provides several commands for compiling, loading, and adding files to user modules. Compiling is used rarely because *F*LORA-*2*'s files get compiled automatically during loading or adding. Loading erases the existing contents of a module and replaces it with the contents of a file. Adding is described in Section 17.6. It is similar to loading, but it is cumulative and does not override the existing contents of a module.

Compilation. The command

```
?- compile{myfile>>mymodule}.
```

generates the byte code for the program to be loaded into the user module named `module`. In practice this means that the compiler generates files named *myfile_mymodule.P* and *myfile_mymodule.xwam* with symbols appropriately renamed to avoid clashes.

If no module is specified, the command

```
?- compile{myfile}.
```

compiles *myfile*.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:

```
?- [myfile].
or
?- load{myfile}.
```

This command loads the byte code of the program in the file *myfile*.flr into the default user module main. If a compiled byte code of the program for the module main already exists and is newer than the source file, the byte code is used. If *myfile*.flr is newer than the compiled byte code (or if the byte code does not exist), then the source file is recompiled and then loaded.

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

```
?- [myfile >> foomod].
?- load{myfile >> foomod}.
```

This loads the byte code of the FLORA-*2* program *myfile*.flr into the user module named foomod. As with the previous form of that command, if a compiled byte code of the program for the module foomod already exists and is newer than the source file, the byte code is used. If *myfile*.flr is newer than that compiled byte code (or if the byte code does not exist), then the source file is recompiled and then loaded.

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:

```
?- [mykb1, mykb2]
```

will load both mykb1 and mykb2 into the default module main. However, simply loading several knowledge bases into the same module is not very useful: the content of the last file 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:

```
?- ['mykb1.flr', mykb2 >> foomod].
```

which loads mykb1.flr into the module main and mykb2.flr into the module foomod.

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 `mykb.flr` exists. If so, the file assumed to contain FLORA-*2* code and is compiled as such. If `mykb.flr` is not found, but *mykb*.`P` or *mykb*.`xwam` 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 you might want to load it, if not). To find out which modules are loaded at the present time, use the primitive `isloaded{...}`. 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.

```
?- isloaded{foo}.
?- ?L= setof{?X|isloaded{?X}}.
```

`setof` and other aggregate operators are discussed in Section 30.

One can also check which files are loaded in what modules and in what mode. The mode is `load` or `add`, and the file names are absolute path names. For instance,

```
?- isloaded{?F,?Module,?Mode}.
?F = /a/b/c/foo.flr
?Module = bar
?Mode = load

?F = /a/b/foo2.flr
?Module = bar
?Mode = add

?F = /a/b/d/bar.flr
?Module = main
?Mode = add
```

There is also a four-place version of `isloaded`: `isloaded{FileAbsName,Module,FileLocalName,Mode}`. The difference is that argument 3 is now the local version of the file name (without the directory part). This version is useful in many ways. For instance, it lets one find the full name of the current file. To do so, recall that FLORA-*2* has a quasi-variable `\@F`, which represents the *local* file name. So, the query

```
?- isloaded{?FullFile,\@,\@F,?}.
```

will bind `?FullFile` to the absolute name of the file where the above query occurs.

Scratchpad code. In some cases—primarily for testing—it is convenient to be able to type up and load small excerpts of code into a running FLORA-*2* session. To this end, the system provides special idioms, `[]`, `[>>module]`, `[+]`, and `[+>>module]`. This causes FLORA-*2* to start reading input 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.

17.6 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

```
?- [+mykb >> foomod].
or
?- add{mykb >> foomod}.
```

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

```
?- [+mykb].
?- add{mykb}.
```

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

```
?- [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:

```
?- compileadd{foo}.
?- compileadd{foo >> bar}.
```

17.7 Module References in Rule Heads

Explicit module references, like *something@module* is not allowed in rule heads or facts because such a practice negates the very idea of a module. Indeed, by definition, the contents of a module is encapsulated and must be defined in the module itself. If something like

```
foo(1,2)@bar.
```

would be allowed in a file like `myfile.flr` that is loaded into a module other than `bar`, this would mean that some other module defines part of the contents of module `bar`—violating module encapsulation. If, on the other hand, `myfile.flr` were added to module `bar` then one should simply write `foo(1,2)@bar` and that file would be loaded into `bar`.

To be sure, FLORA-*2* does allow modules to change the contents of other modules, but this is subject to the rules of encapsulation described in Section 17.13.

FLORA-*2* does allow explicit module references in the arguments of the literals in rule heads. For instance,

```
believes(John, ${flat(Earth)@scifi}).
```

Note that `${flat(Earth)@scifi}` is reified here. This is because `flat(Earth)@scifi` is a truth-valued formula, while arguments must be objects. Reification turns formulas to objects (Section 19.2).

Finally, FLORA-*2* allows module references in rule heads in various meta-query primitives, like `clause{...}`—see Section 29.

17.8 Calling Prolog Subgoals from $\mathcal{F}$LORA-2

Prolog predicates can be called from $\mathcal{F}$LORA-2 through the $\mathcal{F}$LORA-2 module system $\mathcal{F}$LORA-2 models Prolog programs as collections of static *Prolog modules*, i.e., from $\mathcal{F}$LORA-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:

```
?- predicate@\prolog(module)
```

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

```
?- 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 a Prolog program is loaded, the corresponding predicates become available in the default Prolog module. In XSB, the default module is called `usermod` and $\mathcal{F}$LORA-2 can call such predicates as follows:

```
?- foo(?X)@\prolog(usermod).
```

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

```
?- ?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 $\mathcal{F}$LORA-2 with particular ease:

```
?- 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,

```
?- 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).
```

FLORA-*2* provides a special primitive, `p2h{...,...}`, which converts terms to and from the HiLog representation, and the knowledge engineer can use it in conjunction with `@\prolog` to achieve a greater degree of control over argument conversion. This issue is further discussed in Section 19.4.

Builtin Prolog predicates. There is a large number of low-level builtin predicates in the underlying Prolog system. Most of these builtins have been lifted to the FLORA-2 level in various ways and made easier to use. Other predicates were not lifted because they were subsumed by various FLORA-2's constructs. Nevertheless, there still are XSB predicates whose lifting is not being planned because of their rare use, but which can still be useful in some cases. A partial list of such predicates appears in Section 50.

17.9 Calling FLORA-2 from Prolog

Since Prolog does not understand FLORA-2 syntax, it can call only predicates (not frames) defined in FLORA-2 knowledge bases. To expose such predicates to Prolog, they must be imported by the Prolog program.

17.9.1 Importing FLORA-2 Predicates into Prolog

To import a FLORA-2 predicate into a Prolog shell, the following must be done:

- The query

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

  must be executed first. If you are importing FLORA-2 predicates into a Prolog program, say `test.P`, and not just into a Prolog shell then the above must be executed *before compiling or loading* `test.P` and the following additional directive must appear near the top of `test.P`, *prior to any call* to FLORA-2 predicates:

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

- 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 that a double-backslash is required in front of `flimport` because in Prolog the backslash is an escape character. This is also the reason why the backslash appears twice in the various commands in the rest of this section.

Note: You must let Prolog know the location of your installation of FLORA-*2*. This is done by executing the prolog instruction `asserta(library_directory(path-to-flora))`. For instance

```
?- add_lib_dir(a('/home/me/flora2')).
```

before calling any of the FLORA-*2* modules. Observe that `asserta` and *not* `assert` must be used.

The first form for `'\\flimport'` above is used to both import a predicate and also to load the file containing it into a given FLORA-*2* user module. The second syntax is used when the FLORA-*2* knowledge base 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 names start 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 this 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* knowledge bases 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 knowledge base 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 FLORA-*2* user module has been loaded using the following call:

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

Note that in Prolog the `'\\isloaded'` predicate must be quoted and the backslash doubled.

Note: 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 FLORA-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 FLORA-2 knowledge base, then the best way is to start XSB and FLORA-2 using the `runflora` script (page 4) located in the distribution of FLORA-2. □

17.9.2 Passing Arbitrary Queries to FLORA-2

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

To enable such unrestricted queries, FLORA-2 provides a special predicate, `flora_query/5`, which is called from Prolog and takes the following arguments:

- *String*: A string that contains a FLORA-2 query. It can be an atomic frame (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 `["?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 a Prolog atom). Var is a *Prolog* (not FLORA-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 that contains various indicators. The most important ones are `success` and `failure`.
- *XWamState*: This tells whether the returned answer has the truth value "true" (when this argument is bound to 0) or "undefined" (otherwise).
- *Exception*: If the execution of the query is successful, this variable is bound to `normal`. Otherwise, it will be bound to an exception-term returned by XSB (see the XSB manual, if you need to process exceptions in sophisticated ways).

In order to use the `flora_query/5` predicate from within Prolog, the following steps are necessary:

1. The $\mathcal{F}$LORA-*2* installation directory must be added to the XSB search path:

   ```
   ?- add_lib_dir(a('/home/myHomeDir/flora2')).
   ```

2. The query

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

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

3. `flora_query/5` 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.
?- add_lib_dir(a('/home/myHomeDir/flora2')),
   [flora2],
   bootstrap_flora.
:- import flora_query/5 from flora2.
:- import '\\load'/1 from flora2.

?- '\\load'(flrtest).

?- Str='?X[b->?Y].',
   flora_query(Str,['?X'=YYY,'?Y'=PPP], _Status,_XWamState,_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 using a fail-loop, as usual in Prolog. For instance,

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

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,_XWamState,_Exception).
```

yields the same result as

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

is noticeably faster than

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

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

17.10 *FLORA-2* System Modules

FLORA-2 provides a special set of modules that are *preloaded* with useful utilities, such as data type manipulation or I/O. These modules have special syntax, `\modname`, and cannot be loaded by the user. For this reason, these modules are called *FLORA-2 system modules*. For instance, to write to the standard output one can use

```
?- write(Something)@\io.
```

For more details on the currently existing *FLORA-2* system modules see Section 46.

17.11 Including Files into *FLORA-2* Knowledge Bases

The last and the simplest way to construct multi-file *FLORA-2* knowledge bases is via 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 that the file names must be enclosed in *double* quotes.

Unix-style path names (i.e., using forward slashes) are understood universally and irrespective of the actual OS under which FLORA-*2* runs. So, for portability, it is recommended to use only Unix-style path names relative to the directory of the host file that contains the include-directive (e.g., `"../abc/cde.foo"`).

If one does need to use Windows-style path names for some reason, keep in mind that backslashes in path names must be doubled. For instance,

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

17.12 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 quasi-constant `\@`). 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 FLORA-*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 callback functions in C/C++, which allows the callee to behave differently for different clients.

17.13 Module Encapsulation

So far, in multi-module knowledge bases, any module could invoke any method or predicate in any other module. That is, modules were not encapsulated. However, FLORA-*2* lets the user encapsulate any module and export the methods and predicates that other modules are allowed to call. Calling an unexported method or predicate 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 have the same form 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{p(?,?)}.
:- export{?[a(?) -> ?]}.
:- 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 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 the last three export statements above use 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 frames 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 F-logic frames that involve 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.

17.14 Importing Modules

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

FLORA-2 supports import of entire 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:

```
?- [mykb>>foo].
:- importmodule{foo}.

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

This module loads the contents of the file `mykb.flr` into a module `foo` and then imports that module. The importing module itself contains a fact and a query.

Suppose `mykb.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* `mykb.flr` instead of importing it, i.e., if it looked like this:

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

then the same query would have returned two answers: `?X = 123` and `?X = abc`. This is because the latter 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 `mykb.flr` is included then the resulting knowledge base has two `p`-facts and two answers are returned.

Note: If module `modA` imports module `modB` then `modB` must be loaded before any query is issued against `modA`. Otherwise, `modA` might attempt to query `modB` and an error will be issued telling the user that `modB` is not loaded. A good practice to avoid this sort of errors is to load `modB` before `modA`.

17.15 Persistent Modules

Normally, the data in a FLORA-2 module is *transient* — it is lost as soon as the system terminates. The FLORA-2 package `persistentmodules` allows one to make FLORA-2 modules *persistent*. This package is described in the *Guide to* FLORA-2 *Packages*.

18 Truth about Relative File Names and Current Directories

When loading, adding, or opening files, it is often convenient to use relative names. In multifile applications, the use of relative file names becomes not just a matter of convenience, but of necessity because absolute file names make applications non-portable. To properly use relative names in FLORA-2, one must understand how the *current directory* is determined in various situations. The present section explains this and related issues.

Absolute file names (the ones starting with `'/'` in Linux and Mac and those starting with `\` in Windows) do not depend on the location of the FLORA-2 file in which they appear and so they can be used in load/add commands and in I/O operations without the fear of ambiguity. Not so with *relative* file names like `foo/bar.flr` or `../moo.flr`. These depend on the *current directory* in which FLORA-2 executes at the moment when these files are being used. So, what is this "current directory?" This can be summarized as follows:

- When FLORA-2 starts, the current directory is the directory in which FLORA-2 was started. Namely,
  - If FLORA-2 was started from a command window then the current directory of FLORA-2 is the current directory of that window unless that directory is non-writable or is FLORA-2's installation directory. In the latter cases, the current directory is user's home directory.
  - If it was started by clicking on an icon (via ERGO Studio) then the current directory is the user's home directory.

- During loading or adding of an $\mathcal{F}$LORA-*2*'s file, the current directory is that file's directory. If, during loading, that file loads another file, the current directory temporarily is switched to the directory of that latter file.
- When file loading is finished, the current directory returns to what it was before the loading.
- The current directory of $\mathcal{F}$LORA-*2* can be changed at run time by executing the subgoal `File[chdir(`*myNewDir*`)]@\io` — see Section 46.1.

Now, regarding the relative file names that occur in the `load{...}` and `add{...}` commands, the following natural rule holds:

- Any relative file name used in the `load{...}` or `add{...}` command is always relative to the directory of the file in which that load/add command occurs.

This behavior is desirable because it greatly facilitates creation of location-independent multi-file knowledge bases.

The other common place where file names occur are I/O operations, like `tell(myfile)`, which opens `myfile` for writing (see Section 46.1 for more details on I/O). Here the situation is more subtle because I/O commands can execute in many different contexts, not necessarily in the context of the file in which they are physically located. For that reason, the rule is that

- any relative file name that occurs in an I/O command is always relative to the directory that is current at the time that command is executed.

For example, we know from the above that during loading of a file the current directory is the directory of that file. Therefore, if a query like

```
?- tell(abc)@\io, writeln(abcdefg)@\io, told@\io.
```

occurs in a file `/foo/bar/moo/myfile.flr` then, when that file will be loaded, the file `abc` will be created in the directory `/foo/bar/moo/` and its contents will be a single line `abcdefg`. This is because, as we discussed, the current directory during loading of a file is that file's directory. In contrast, if our file has a rule

```
makefile(?X) :- tell(?X)@\io, writeln(12345678)@\io, told@\io.
```

and `makefile(cde)` is called at some point, this will create a file, called `cde` and containing the single line `12345678`, in the directory that was current at the time `tell(?X)@\io` (and `makefile(cde)`) was called.

Thus, the context in which files are interpreted in I/O commands is not uniform and depends on how these commands are called—a somewhat unpleasant situation. Fortunately, to rectify this problem, FLORA-2 provides a primitive, `here{...}`, which takes any subgoal and calls it in the directory of the file in which `here{...}` occurs. For instance, with a slightly modified rule

```
makefile(?X) :- here{tell(?X)@\io}, writeln(12345678)@\io, told@\io.
```

a call `makefile(cde)` will create `cde` in the directory of our example file, i.e. `/foo/bar/moo/`, and not in the directory where `makefile(cde)` was called.

Finally, we note that `here{...}` can take any goal, even a complex one, provided it is enclosed in parentheses. For instance, `here{(p(?X,?Y),q(?Y))}`. In fact, this primitive can be used to temporarily localize the context of execution for *any* subgoal, not only for I/O commands.

19 HiLog and Meta-programming

HiLog [5] is the default syntax that FLORA-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 base 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 [5]. 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 [5].

Any Id-term in FLORA-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 [5], and its implementation in FLORA-2. In HiLog, functor terms that appear as arguments to predicates and the base formulas

(i.e., predicates that are applied to some arguments) belong to the same domain. In contrast, in FLORA-*2* they are in different domains.[11] 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 19.1; it binds `?X` to the base formula `a(b)` in the current module. The answer to this query is 'No' because `?X` is bound to the base formula `a(b)`, while `a(b)` in `p(a(b))` is a HiLog term.

Our earlier query, (3), will also not work (unlike 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`.

```
?- p(?X), ?X(p), ?X@ \@.
```

Like in classical logic, `foo` and `foo()` are different terms. However, it is convenient to make these terms synonymous 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 `flapply(foo)` is different from `flapply(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:

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

In the following,

```
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. Making `p` and `p()` synonymous does not make them synonymous with `p()()`—the latter is distinct from both `p` and `p()` not only as a term but also as a formula. Thus, in the following, all queries fail:

[11] This is allowed in *sorted HiLog* [4].

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

19.1 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

?- arg(2,f(?X,3)@\prolog,?Y)@\prolog.
?Y = 3
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*, a variable can be bound to either a formula or a term. 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 frame 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, `~`. This operator treats its arguments as formulas and unifies them as such. For instance, `?X ~ a[m->v,k->?V]` binds `?X` to the frame `a[m->v,k->?V]` and `a[m->v,k->?V] ~ ?X[?M->v,k->p]` unifies the two frames by binding `?X` to `a`, `?M` to `m`, and `?V` to `p`.

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

```
?- ?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

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

```
?- ?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:

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

Meta-disunification. The negation of meta-unification is *meta-disunification*, `!~`. For instance,

```
?- abc !~ neg cde.

Yes
```

Note that `neg`, `\naf`, and `\+` tie stronger than `~` and `!~`, so the above is parsed as `abc!~neg(cde)` and `neg cde!~abc` is parsed as `neg(cde)!~abc` — *not* as `neg(cde!~abc)`.[12]

19.2 Reification

It is sometimes useful to be able to treat FLORA-*2* frames and predicates as objects. For instance, consider the following statement:

```
Tom[believes-> Alice[thinks->Flora2:coolThing]].
```

The intended meaning here is that one of Tom's beliefs is that Alice thinks that Flora2 is a cool thing. However, minute's reflection shows that the above has a different meaning:

```
Tom[believes-> Alice].
Alice[thinks->Flora2:coolThing].
```

That is, Tom believes in Alice and Alice thinks that Flora2 is cool. This is different from what we originally intended. For instance, we did not want to say that Alice likes FLORA-*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->Flora2: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 Flora2 is a cool thing one should write:

[12] This is in contrast to `=`, `==`, `<`, etc., which bind stronger than `neg`, `\naf`, and `\+`. For instance, `neg cde=abc` is parsed as `neg(cde=abc)`. This difference in parsing makes sense because `~` and `!~` expect formulas as its operands, while `=`, `==`, and other comparison operators expect terms.

```
Tom[believes-> ${Alice[thinks->Flora2: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 because modules are unbound:

```
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 [21].

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, \naf 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 frames, 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.[13]

[13] 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.

Note that rule Ids and other meta-data (see Section 36) can be supplied with reified rules just like it can be with regular rules. For instance,

```
... :- ?X=${@!{abc[tag->foo]} head(?X):-body(?X,?Y)}.
```

Reification and meta-unification. Reification should not be confused with meta-unification, although they are closely related 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,

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

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

```
?- ${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`).

19.3 Meta-decomposition

FLORA-*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,

```
?- ?X=p(a,?Z)@\prolog, ?X=..?Y.

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

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

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

```
?- 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,

```
?- ${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,

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

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

For frame literals, the head of the list has the form `flogic(FrameSymbol,Module)`. The `FrameSymbol` argument represents the type of the frame and can be one of the following: `->`, `*->`, `=>`, `*=>`, `+>>`, `*+>>`, `->->`, `*->->`, `:`, `::`, `boolean` (tabled Boolean methods), `*boolean` (class-level tabled Boolean methods), `%boolean` (transactional, nontabled Boolean methods), `:=:`, `[]` (empty frames, such as `a[]`). Here are a number of examples that illustrate the use of `=..` for decomposition of frames:

```
${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 supports explicit negation (Section 20.5). The corresponding type designators are `neg_hilog` and `neg_flogic`. For instance,

```
${\neg a[b->c]@foo} =.. [negation(neg), $a[b->c]@main]
${\neg p(a,b)@foo} =.. [negation(neg), $p(a,b)@main]
${\naf a[b->c]@foo} =.. [negation(naf), $a[b->c]@main]
${\naf p(a,b)@foo} =.. [negation(naf), $p(a,b)@main]
```

Additional examples:

```
${foo;bar} =.. [logic(or),${foo},${bar}]
${\if foo\then bar\else moo} =.. [control(ifthenelse),main,${foo},${bar},${moo}]
${\while foo \do bar} =..[control(whiledo),main,${foo},${bar}]
${insert{?V|p(?V)}} =.. [update(insert),[?V],${p(?V)}]
${deleteall{?V|p(?V)}} =.. [update(deleteall),[?A],${p(?A)}]
${?X=min{?V|p(?V)}} =.. [logic(and),${?T=min{?V|p(?V)}},?X=?T]
avg{?V[?T]|p(?V,?T)} =.. [aggregate(avg),?V,[?T],${p(?V,?T)},?Result]
count{?V[?G]|p(?V,?G)} =.. [aggregate(count),?V,[?G],${p(?C,?G)},?R]
${insert{p,(q:-r)}} =.. [update(insert),main,[${p},${q :- r}]].
${delete{p,(q:-r)}} =.. [update(delete),main,[${p},${q :- r}]].
${wish(nonvar(?X))^p(?X)} =.. [quantifier(delay),wish,${nonvar(?X)@\plg},${p(?X)}].
```

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

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

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

However, that last feature, while it works for most statements, is *not* fully implemented in the sense that the terms produced (if produced at all) on the left-hand side might not be a valid *F*LORA-*2* term or formula or the internal representations may be slightly different.

The statements for which the reverse mode of `?Variable =.. ?List` has not yet been implemented include `clause{...}`, `newmodule{...}`, and some others. The statements for which `=..` is not fully reversible are the update operators, statements, and a few others. For instance, while

```
?- ${insert{p,(q:-r)}} =.. [update(insert),main,[${p}, ${q :- r}]].
```

succeeds, the following will fail:

```
?- ${delete{p,(q:-r)}} =.. [update(delete),main,[${p}, ${q :- r}]].
```

because the internal representations of the two sides of `=..` happen to be slightly different. Nevertheless, `=..` does produce valid insert- and delete-statements from appropriate list on the right side. For instance,

```
?- ?I =.. [update(insert),main,[${p}, ${r}, ${q :- r}]], ?I.
?- p,q,r.
```

will succeed, i.e., a valid insert statement was constructed in `?I` and the call `?I` did perform the insertion. Likewise,

```
?- ?D =.. [update(delete),main,[${p}, ${r}, ${q :- r}]], ?D.
```

constructs a proper delete statement for `?D` and a call to `?D` then performs the deletion.

The low-level Prolog `=..` is also available using the idiom `(?Term =.. ?List)@\prolog`. This is rarely needed, however. One might use this when the term to be decomposed is known to be a Prolog term (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).

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

The native HiLog support in $\mathcal{F}$LORA-*2* causes some tension when crossing the boundaries from one system to another. The reason is that $\mathcal{F}$LORA-*2* terms and Prolog terms have different internal representations. 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, FLORA-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 FLORA-2 shell, they both succeed:

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

but the following will fail:

```
?- p2h{f(a),?X}, p2h{?X,?X}.
?- 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 the terms that appear in a FLORA-2 knowledge base (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 the 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 46, which provides the needed conversions for the I/O predicates.

Another mechanism for calling Prolog modules, described in Section 17.8, is 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 representations. However, as mentioned above, such conversion is sometimes not necessary and the use of `@\prologall` might incur unnecessary overhead.

20 Negation

FLORA-2 supports three kinds of negation: a *Prolog-style negation* `\+` [6]; *default negation* based on *well-founded semantics*, denoted `\naf`, [14, 15]; and *explicit negation* `\neg`, which is analogous to what is called "classical" negation in [8].

These three types of negation are quite different and should not be confused. Prolog-style negation does not have a model-theoretic semantics and it is unsatisfactory in many other respects. It is included in FLORA-2 for completeness and is primarily used when one needs to negate a Prolog predicate (in which case it is much faster than default negation). Explicit negation is mostly syntactic sugar that enables one to represent negative information explicitly. Default negation is a logically sound version of Prolog-style negation.

20.1 Default Negation `\naf` vs. Prolog Negation `\+`

FLORA-2 has *three* operations for negation: `\naf`, `\+`, and `\neg`. In this subsection we discuss the first two and Section 20.5 describes the third.

Prolog negation is specified using the operator `\+`. Negation based on the well-founded semantics is specified using the operator `\naf`. The well-founded negation, `\naf`, applies to predicates that are tabled (i.e., predicates that do not have the `%` prefix to be discussed in detail in Section 24) or to frames that do not contain transactional methods (i.e., methods prefixed with a `%`).

The semantics for Prolog negation is simple. To find out whether `\+ Goal` is true, the system first asks the query `?- Goal`. If the query fails then `\+ Goal` 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 `\+ Goal` 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. Prolog negation is the recommended type of negation for non-tabled predicates (but caution is advised).

The well-founded negation, `\naf`, 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 we have the rule `p :- \naf p` then the truth value of `p` is *undefined*. Although the details of this semantics are somewhat involved [15], 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 be applied to goals that consist entirely of tabled predicates or frames. Although *FLORA-2* allows `\naf` to be applied to non-tabled goals, this may lead to unexpected results. For instance, Section 27 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 `\naf` can be used as operators inside and outside of the frames. For instance,

```
?- \naf %p(a).
?- \+ %p(a).
?- \naf X[foo->bar, bar->foo].
?- X[\naf foo->bar, bar->foo, \+ %p(?Y)].
```

are all legal queries. Note that `\+` applies only to non-tabled constructs, such as non-tabled *FLORA-2* predicates and transactional methods.

To apply negation to multiple formulas, simply enclose them in parentheses. (Parentheses are not needed for singleton formulas used in earlier examples.)

```
?- \+ (%p(a),%q(?X)).
?- \naf (p(a),q(?X)).
?- \naf (?X[foo->bar], ?X[bar->foo]).
```

20.2 Default Negation for Non-ground Subgoals

One major difference with other implementations of the well-founded default negation is that *FLORA-2* lets one apply it to formulas that contain variables. Normally, systems either require that the formula under `\naf` is ground or they interpret something like `\naf p(?X)` as meaning "not exists `?X` such that `p(?X)` is true" — the so-called *Not-Exists* semantics. However, this is not the right semantics in many cases. The right semantics is usually "there exists `?X` such that `p(?X)` is *not* true." This semantics is known as the *Exists-Not* semantics. Indeed, the standard convention for variables that occur in the rule body but not in the head is existential. For instance, if `?X` does not occur in the head of some rule, `p(?X)` in the body of that rule is interpreted as ∃`?X` `p(?X)`. Negation should *not* be treated differently, i.e., `\naf p(?X)` should be interpreted as ∃`?X` `\naf p(?X)`. Worse yet, if `?X` *does* occur in the rule head then it is confusing and error-prone to interpret `h(?X) :- \naf p(?X)` as `h(?X) :- \naf` ∃`?X p(?X)` using the Naf-Exists semantics. And without the Naf-Exists semantics, `\naf p(?X)` has no meaning, if `?X` happens to be non-ground.

*F*LORA-*2* takes a different approach. For body-only variables that appear under `\naf` the semantics is Exists-Not. In addition, `\neg p` implies `\naf p`. So,

```
\neg p({1,2}).
?- \naf p(?X).
```

returns the bindings 1 and 2 for `?X`. For variables that occur both under `\naf` *and* in the rule head, the semantics is also standard: universal quantification that applies to the entire rule. What happens if `?X` is not ground at the time of the call to `p(?X)` (whether `?X` does or does not occur in the rule head)? In that case, *F*LORA-*2* *defers* the call in the hope that `?X` might become ground later. For instance, in

```
p(2).
?- \naf p(?X), ?X=1.
```

the query succeeds because the call `\naf p(?X)` will be delayed past the moment when `?X` becomes ground. Finally, what happens if `?X` does not become ground even after the delay? Still the Exists-Not semantics is used. If `p(?X)` is false for all `?X` (i.e., if `\naf` ∃`?X p(?X)` holds) then, in particular, ∃ `\naf?X p(?X)` is also true (assuming an infinite number of constants) and the query succeeds with the truth value *true*. If ∀`?X p(?X)` is true then `\naf` ∃`?X p(?X)` is false. However, if none of the above cases apply then we are in a gray area and there is not enough information to tell whether `\naf` ∃`?X p(?X)` is true or false, so this subgoal succeeds with the truth value *undefined*. For instance, suppose that the only facts in the KB are the ones below:

```
p({1,2}).
?- \naf p(?X).
?- \naf q(?X).
```

Then the first query succeeds with the truth value *undefined*, since `\naf` ∃`?X p(?X)` is not true. On the other hand, the second query succeeds with the truth value *true*. In both cases, *F*LORA-*2* makes an open domain assumption by refusing to commit to true or false based only on the explicitly known elements of the domain of discourse.

Finally, what if the user does want the Not-Exists semantics after all? In *F*LORA-*2* one must then say this explicitly and in a natural way through the use of existential quantifier:

```
?- \naf exists(?X)^p(?X).
```

More on the logical quantifiers is given in Section 21.

20.3 Non-ground Subgoals Under \+

When \+ is applied to a non-ground goal, the semantic is the standard Prolog's one: If for *some* values of 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 ∀?`Vars` ¬ `Goal`, where `?Vars` represents all the nonbound variables in `Goal`. However, here ¬ should be understood not as classical negation but rather as a statement that `Goal` cannot be proved to be true.

20.4 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 mean. Is undefinedness covered by 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] :- \naf e[f->g].

?- true{a[b->c]}.

Yes

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

Yes

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

Yes
```

It should be noted that the primitives `true{...}` and `undefined{...}` can be used only in top-level queries or in the rules whose heads are not mutually recursive with any of the query literals. Otherwise, the result is undefined. The expression `false{Goal}` is equivalent to `\naf Goal`, and can be used anywhere.

In addition, sometimes it may be necessary to to check if a succeeding query succeeds with the

truth value *true* or *undefined*. (A failing query never succeeds, so there is nothing to test in that case.) To this end, FLORA-*2* provides the primitive `truthvalue` which takes a variable that gets bound to the truth value of the query. For instance, suppose that the query `q(?X)` has two answers: `?X=a` is a true answer and `?X=b` is undefined (not false!). Then

```
q(a).
q(b) :- \undefined.
?- q(?X), truthvalue{?_P},
   \if ?_P == \true \then writeln(?X=true)@\io
   \else writeln(?X=undefined)@\io.
b = undefined
a = true

?X = a
?X = b  - undefined
```

As with `true{...}` and `undefined{...}`, the `truthvalue{...}` primitive can be used only in top level queries or in rules that are not mutually recursive with any of the query literals.

20.5 Explicit Negation

Explicit negation is denoted using the connective `\neg`, for instance, `\neg p`. It is a weaker form of classical negation and is quite unlike the default negation. First, explicit negation can appear both in rule heads and rule bodies. Second, unlike the default negation or Prolog negation, in order to conclude `\neg p` one must actually *prove* `\neg p`—*not* simply fail to prove `p`. In other words, establishing `\neg p` is a harder requirement than establishing `\naf p`, and `\neg p` *always* implies `\naf p`.

Also, unlike classical negation, the law of excluded middle does not hold for `\neg`, so both `p` and `\neg p`. In contrast, `p` and `\naf p` cannot be both true and cannot be both false—it is always the case that one is true and the other is false. Can both `p` and `\neg p` be true? The answer may surprise: *yes*. FLORA-*2* does not check that by default because this is somewhat expensive. To tell it to check this inconsistency, two things must be done:

1. The module where these facts occur must be declared as defeasible using the
   `:- use_argumentation_theory.`
   directive—see Section 38 and
2. The facts and the rules where the above literals occur in the rule heads must be made defeasible (also see Section 38). This can be done either by giving these rules/facts the defeasible property or by assigning them explicit defeasibility tags:

For example,

```
:- use_defeasibility_theory.
@@{defeasible} p.    // fact has defeasibility property
@{abc} \neg p.       // explicit defeasibility tag
?- p.
No
?- \neg p.
No
```

As you can see, in this case, FLORA-2 will detect an inconsistency and will "defeat" the offending inferences by making both of them false. One can also find out why the various inferences were defeated—see Section 38.

The explicit negation connective, `\neg`, can be applied to conjunctions or disjunctions of literals. However, `\neg p,q` is not allowed in the rule head, since this is tantamount to a disjunction, `\neg p \or \neg q`. In a rule body, the idioms `\neg\naf p` and `\neg \+ p` are illegal, but `\naf\neg p` and `\+\neg p` are legal and should be informally understood as statements that `p` is not known to be false.

As with the default negation `\naf`, `\neg` is allowed inside the frames in front of `->` and `=>` and in front of Boolean (non-transactional) methods. For instance,

```
a[\neg b->c], a[|\neg b => c|], a[\neg b], a[|\neg b|]
```

are allowed, but `a[\neg %b]`, `a[\neg b ->-> c]`, `a[\neg b +> c]` are illegal.

At present, explicit negation works with defeasible reasoning, but it is not treated in any special way in modules that do not use defeasible reasoning. For instance, no consistency check is done to ensure that `p` and `\neg p` are not true at the same time. This is future work.

Explicit negation is fully integrated with meta-programming. For instance, the following is valid syntax:

```
?- ?X = ${a[b->c]}, \neg ?X.
```

This is equivalent to `?- \neg a[b->c].` The meta-unification and meta-decomposition operators, $\sim$ and `=..`, introduced in Sections 19.1 and 19.3, are also aware of explicit negation. For instance,

```
?- ${\neg a[b->c]@foo} =.. [neg_flogic(->,foo), a, b, c].
?- \neg a[b->?C]@foo ~ \neg ?X@?Z, \neg ?X ~ ${\neg ?A[?B->cc]@foo}.
```

The second query produces the following answer:

```
?C = cc
?X = ${\neg \neg a[b -> cc]@foo}
?Z = foo
?A = a
?B = b
```

The answer for `?X` might look a little strange, but double negation works as expected:

```
?- ${\neg \neg a[b -> cc]@foo} ~ ${a[b -> cc]@foo}.
```

Transactional HiLog literals and methods cannot be negated using `\neg`. For instance, the following literals are syntax errors:

```
\neg %p
\neg a[%p]
```

Similarly, $\mathcal{F}$LORA-*2* update or aggregate operations cannot be negated.

21 General Formulas in Rule Bodies

Unlike most other rule engines, $\mathcal{F}$LORA-*2* supports a much larger variety of formulas in the rule body. In due time we will see various if-then-else clauses, loops, etc. But the free use of logical quantifiers `forall` (`all`, `each` are also accepted) and `exists` (`exist`, `some` are also accepted) is, perhaps the most unique. In fact, the rule bodies in $\mathcal{F}$LORA-*2* can have arbitrary formulas involving quantifiers, `\and`, `\or`, `\naf`, and the logical implications `~~>`, `<~~`, `<~~>`, `==>`, `<==`, and `<==>`. Here are some examples:

```
h1(?Z) :- \naf exists(?X)^(\naf ((p(?X) ~~> q(?X,?Z)))).
h2(?Y) :- forall(?X)^(p(?X,?Y) ~~> q(?X)).
h3(?V,?W) :- forall(?Y,?Z)^(\naf exists(?X)^(pp(?X,?W,?Y,?Z), \naf qq(?X,?Y,?V,?Z))).
?- forall(?X)^(p(?X)<~~>p(?X)).
```

Other connectives, such as `==>`, `<==`, `<==>`, `\if-\then-\else`, etc., can also be used in conjunction with the quantifiers. The definitions of all these implications are as follows:

- ϕ`~~>`ψ (and `\if` ϕ `\then` ψ) is a shorthand for `\naf` ϕ `\or` ψ.
  ψ`<~~`ϕ is the same as ϕ`~~>`ψ, while ϕ`<~~>`ψ is (ϕ`~~>`ψ `\and` ϕ`<~~`ψ).
- ϕ`==>`ψ is a shorthand for `\neg` ϕ `\or` ψ.
  ψ`<==`ϕ is the same as ϕ`==>`ψ, while ϕ`<==>`ψ is (ϕ`==>`ψ `\and` ϕ`<==`ψ).

- `if` ϕ `then` ψ `else` η is $(\phi$`~~>`ψ `\and` $($`\naf` $\phi)$`~~>`$\eta)$.

It should be noted that *generally* it makes no sense to place non-logical operators in the scope of a quantifier or `\naf`. For instance,

```
?- \naf load{abc}.
```

does not make a whole lot of sense (try it!). But some uses of quantification in conjunction with actions do make sense and are quite natural: For instance,

```
?- forall(?X)^(data(?X)~~>writeln(?X)@\io).
?- some(?X)^(data(?X)~~>writeln(?X)@\io).
```

will print out the contents of the predicate `data/1` or just one value (in the second case).

Quantifiers in $\mathcal{F}$LORA-*2* are implemented through an extension of the well-known Lloyd-Topor transformation [13] except that the role of negation here is played by `\naf`, not `\+`. However, unlike Prolog for which the Lloyd-Topor transform was originally defined, $\mathcal{F}$LORA-*2* gives a logical treatment to negation of non-ground formulas. In particular, as discussed in Section 20.2, when negating non-ground subgoals the truth value may be *undefined*. The universal quantifier and `~~>` have implicit negation in them, so, in order to know when to anticipate undefinedness, it is useful to understand what it means to have a non-ground subgoal under `\naf`.

We say that a variable in the body of a rule occurs *positively* if it occurs in the scope of an even number of `\naf`s. A variable occurs *negatively* if it occurs in the scope of an odd number of `\naf`s. So, undefinedness can happen if some variable in the body of a rule occurs negatively and does not occur positively.

To make the above more precise, we explain how to count the `\naf`s. This is not immediate because of the implicit negations mentioned earlier. So, for the purpose of counting the `\naf`s, one must break each `forall(Varlist)^`ϕ into (`\naf exists(Varlist)^( \naf` ϕ)). Each ϕ`~~>`ψ must be broken into (`\naf` ϕ `or` ψ), and similarly with `<~~`; `if-then` is synonymous with `~~>`; and `<~~>` is a conjunction of `~~>`, and `<~~`. The connective `\if` ϕ `\then` ψ `\else` η is treated for the purpose of counting `\naf`s as ϕ `\and \naf` ψ `\or \naf` η. Note that `==>` and related connectives do not involve `\naf`s.

21.1 Quantification of Free and Anonymous Variables

In logic, free variables (i.e., variables that do not occur in any quantifier) do not normally make sense because it is unclear how to determine truth of a formula with such a variable. Nevertheless, non-quantified variables are widely used. In fact, languages like Prolog and SQL have no syntax

for quantifiers, so one might be misled into thinking that all variables there are non-quantified. This is incorrect, however: both languages have a set of conventions by which all variables are quantified *implicitly*. In Prolog, for example, the convention is that *all* variables are implicitly quantified outside of the rule, i.e., $\forall X, Y, Z(head(...) : -body(...))$. This rule is often also stated equivalently as follows: variables that occur in the body *only* are quantified existentially in the body and all the rest are universal in the scope of the entire rule, i.e., $\forall X, Y, Z(head(X, Y, Z) : -\exists V, W(body(X, Y, Z, V, W)))$. The situation gets muddies when negation as failure is taken into account and we will not go there right now.

In $\mathcal{F}$LORA-*2*, the situation is a bit more interesting because here we have *explicit* quantifiers, but it is not mandatory to quantify all variables. According to the previous discussion, variables that are not quantified explicitly must be quantified implicitly according to some rules. What are these rules? The implicit quantification rules for free variables make a distinction between *named* variables (which includes *don't-care* variables) and *anonymous* variables. In addition, it matters whether a free variable occurs under the scope of a `\naf` or not:

- *Anonymous variables*:
  First, note that an anonymous variable is never explicitly quantified simply because it has no name (at least, no name the user knows about) and so one does not have anything to refer to that variable in the quantifier (something like `forall(?)` is not allowed because it can be highly ambiguous).
  So, the rule for such variables is that each anonymous variable that occurs in some literal, `p(...,?,...)` or `\neg p(...,?,...)`, is implicitly quantified with `exists` immediately before that literal, i.e., `exists(?var123)^p(...,?var123,...)` or `exists(?var123)^\neg p(...,?var123,...)`, where `var123` is some internal compiler-generated name for that anonymous variable. This existential quantifier takes precedence over `\naf` (but not over `\neg`, as we just saw).

- *Named variables that do not occur under the* `\naf`:
  The rules for these variables are the same as in Prolog: if they occur in the head (and possibly in the body also) then they are universal with respect to the entire rule; if they occur in the body only then they are existential with respect to the body.

- *Named variables under the* `\naf`:
  This implicit quantification rule has a bit of computational flavor and is not completely declarative, which has to do with the fact that the declarative definition for `\naf` exists for top-down computation essentially only for the variable-free case.
  If a named variable gets grounded during the evaluation *before* the evaluation of `\naf` starts then it is not a variable any more and the question is moot. If it remains unbound by the time `\naf` gets computed then the variable is treated existentially *outside* of the scope of `\naf`. In many cases, this situation leads to answers whose truth value is *undefined*. This

because FLORA-2 quantifiers do not specify the domain for the quantified variables and for some domains such queries may be true and for some false.

21.2 The Difference Between ~~>, ==> (or, <~~, <==), and :-

Note that the connectives ==>, <==, and <==> do not involve \naf: they are translated using \neg. For instance, ϕ==>ψ is a shorthand for (\neg ϕ) \or ψ, while ϕ~~>ψ is a shorthand for (\naf ϕ) \or ψ (and \if-\then is synonymous to ~~>). This means that there is a significant semantic difference between these types of formulas:

- ϕ==>ψ is true iff: when ϕ is *not known* to be false (i.e., when \neg ϕ *cannot* be proven true) then ψ is also true.
- ϕ~~>ψ is true iff: when ϕ is true then ψ is also true.

The second statement is true in *more* cases. To see this, recall that ϕ==>ψ is (\neg ϕ) \or ψ while ϕ~~>ψ is (\naf ϕ) \or ψ. Note that (\neg ϕ) implies (\naf ϕ) i.e., the latter is true more often than the former. Therefore, (\naf ϕ) \or ψ is true more often than (\neg ϕ) \or ψ.

Experience shows that in most cases ϕ~~>ψ is the correct intended usage in rule bodies. If the user feels that ==> is needed, a careful analysis of the intended meaning based on above cases is strongly suggested.

The other important differences are syntactic: ϕ==>ψ can appear both in the body of a rule (i.e., on the right of :− and in the head (on the left of :−) provided that neither ϕ nor ψ use the \naf. In contrast, ϕ~~>ψ can be used only in the body of a rule because its very definition involves \naf.

As to :-, this connective is very different from the aforementioned ones, as it is the only one that forms what we call *rules*. What's on the left of :- is the head of the rule and what's on the right is the body. Neither ~~> nor ==> (nor their leftward brethren <~~ and <==) form rules: they can appear only in rule bodies in FLORA-2 (but in $\mathcal{ERGO}$ ==> can also appear in rule heads). But the main difference is that these connectives *test* if, say, ϕ~~>ψ is true (false, or undefined) for some variable bindings in ϕ and ψ. These tests are part of the process of determining if the body of a rule is true (false or undefined). In contrast, *head*:-*body* tests if *body* is true (undefined) and, if so, *derives* that *head* is also true (respectively, undefined). For instance, consider the following example:

```
p(1), q(1), r(2,1).
w(?X) :- r(?X,?Y), (p(?Y) ~~> q(?Y)).
```

Here one can verify that `p(?Y) ~~>q(?Y)` is true when $?Y$ is bound to 1, and it so happens that `r(?X,?Y)` is also true for this binding if `?X` is bound to 2. Then the rule on the second line of the example *derives* the fact `w(2)`. It is important to understand that, in contrast, neither `p(?Y) ~~>q(?Y)` nor `p(?Y) ==>q(?Y)` derive anything.

21.3 Unsafe Negation

Evaluation of `\naf` or `forall` (which translates into `\naf exist(...)^(\naf ...)` and thus also involves negation) may encounter situations where $\mathcal{F}$LORA-*2* is unable to determine the real truth value of an answer and will report it as undefined. When this happens, an *unsafe negation* *\(naf)* warning, such as below, will be issued:

```
++Warning[FLORA-2 ]> File: ..., line: ...
 Subgoal: (\naf (q(?A,?B), (\naf p(?A))))
 Unbound variables: [?B]
 The subgoal has unbound variables under \naf or 'forall'.
 Ergo is unlikely to evaluate this to true or false.
 Try to bind free & quantified variables to finite domains.
                .....
```

Section 43.8.7 explains how to suppress such warnings, *if the query/rule cannot be corrected*, but here we will dwell on the reasons for such warnings and on how queries can be improved by reformulation. Note that in the above case, the variable `?B` in `q/2` will be unbound during the evaluation of the outer `\naf`, whence the warning.

The first reason for having unbound variables during the evaluation of `\naf` is that unquantified, free variable may indeed appear under a `\naf`. For instance,

```
p(123).
?- \naf p(?X).

?X = ?_h10180  - undefined
```

Here `?X` is a free variable under `\naf` and the reason for undefinedness is that for some `?X` the query is true (e.g., for 321) and for some false (e.g., 123). If the query were reformulated by binding `?X` to a specific domain then we would have gotten much more meaningful answers:

```
?- ?X \in 122..124, \naf p(?X).
?X = 122
?X = 124
```

Here we specified the domain as a range `122..124` — see Section 16.

More subtle reasons for the inability to evaluate queries to true or false (and for the appearance of the above warning) is when a universally quantified variable is not bound to a concrete domain. Consider the following set of facts and a query:

```
p(1), p(2).
q(1,a), q(2,a), q(3,a).
q(1,b), q(2,b), q(3,b).
q(1,c).
?- forall(?X)^exists(?Z)^(q(?X,?Z)~~>p(?X)).

Yes - undefined
```

In addition to the undefined truth value, the query will issue a runtime warning of the above kind, complaining about the unbound variable `?X` in `q(...,...)`. If, however, that variable is bound to a finite domain, the warning will disappear and the query will be evaluated to true:

```
?- forall(?X)^(?X \in 1..3 ~~> exists(?Z)^((q(?X,?Z)~~>p(?X)))).
```

Binding to an explicit domain is not always necessary, as sometimes FLORA-2 can determine the domain from the context, as in this query:

```
?- forall(?X)^(q(?X,?Z)~~>p(?X)).
?Z = ?_h5646
```

where FLORA-2 automatically determines that `?X` can take only the values that occur in the first argument of `q(...,...)`. When such determination is not easy to made automatically, user's help with query reformulation could make a difference, as in the above examples.

22 Inheritance of Default Properties and Types

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. This section describes the extensive support for inheritance both for default values as well as for typing information.

FLORA-2 supports two types of inheritance: *structural* and *behavioral*. Structural inheritance applies to signatures only, i.e., to *class* frame formulas that use the `=>`-style arrows. These formulas specify the *type information* for classes as a whole. For instance, if `student::person` holds and we

have the signature `person[|name=>string|]` then the query `?- student[|name=>?X|]` succeeds with `?X=string`.

Behavioral inheritance is much more involved. FLORA-*2* supports two versions of behavioral inheritance—*monotonic* and *non-monotonic*—and the choice can be specified on the per-module basis. In both cases, behavioral inheritance concerns class frame formulas that use the `->`-style arrows or to Boolean class frames, and these formulas are inherited to subclasses and class members. The key difference is that *monotonic* inheritance (both structural and behavioral) is cumulative and resembles the way types are inherited in structural inheritance. In contrast, behavioral inheritance is *non-monotonic* in the sense that the formulas being inherited are understood as default specifications that can be overridden by the information explicitly specified for subclasses. This also implies that adding new information to subclasses may invalidate previously true facts, i.e., true (inferred) information does not necessarily grow monotonically as we add more data.

22.1 Introduction to Inheritance

FLORA-*2* distinguishes between information defined for a class as a whole and information defined for an individual object only. The former, class-wide information is inherited to the members of the class and to its subclasses, and is specified using the frame formulas of the form (note the vertical bars):

```
obj[|Meth->Val|]
obj[|Meth=>Val|]
obj[|BoolProp|]
obj[|=>BoolProp|]
```

These formulas normally occur as part of the specifications for classes. Object-specific information is given using the formulas of the form (no vertical bars!):

```
obj[Meth->Val]
obj[Meth=>Val]
obj[BoolProp]
obj[=>BoolProp]
```

This information is always attached to individual objects. Even if an object represents a class (and in FLORA-*2* classes are also objects), this information does *not* apply to the members of that class or its subclasses. For instance,

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

does not propagate to an object such as `John` even if `John:person` is true. The property `avg_age->40` is likewise not inherited by the various subclasses of `person`, such as `student` or `employee`: it would not make sense to propagate such information because an average age of all persons is likely to be different from the average age of students and employees, and this is why we wrote `person[avg_age->40]` and not `person[|avg_age->40|]`. Similarly, attributes that typically refer to individuals are better specified as object-level information, because normally there is nothing to inherit these attributes to. For instance,

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

makes sense but neither `person[avg->30]` nor `person[|avg->30|]` does.

Class formulas typically define *default* properties for the objects in a class, which are inherited unless there is information to the contrary. For instance, suppose we have

```
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 evidence to the contrary) that `Scottish[|nativeLanguage -> English|]`.

Note that a class formula becomes an object-level formula when its information is inherited to the members of the class. For instance, if we are also told that `John:British` then `John` inherits the native language as follows: `John[nativeLanguage->English]` (no bars! because there is nothing to inherit to from `John`). However, the inherited property remains a class formula when it is inherited to a subclass (e.g., `Scottish[|nativeLanguage->English|]`).

Suppose now that we are told that `John` is actually a native speaker of `Gaelic` via the fact `John[nativeLanguage->Gaelic]`. In that case, the explicitly given property `nativeLanguage -> Gaelic` *overrides* the default property `nativeLanguage->English` and the latter is *not* inherited.

Overriding can happen also at the level of classes. Suppose $\mathcal{F}$LORA-*2* has this pair of facts: `American[|nativeLanguage->English|]` and `Manuel:American`. In the absence of any other information, we would have derived by inheritance that `Manuel[nativeLanguage->English]`. But if our knowledge base also has these facts:

```
PuertoRican::American.
PuertoRican[|nativeLanguage->Spanish|].
Manuel:PuertoRican.
```

then $\mathcal{F}$LORA-*2* would discover that, since `PuertoRican` is a subclass of `American`, the property `nativeLanguage->Spanish` is more specific to `Manuel` than `nativeLanguage->English`, and so,

by inheritance, it would derive `Manuel[nativeLanguage->Spanish]`. Had *FLORA-2* been told that `Manuel[nativeLanguage->Portuguese]`, this most specific information would override that inheritance of `nativeLanguage->Spanish`.

The signature frame formulas are inherited similarly, but there is no overriding. For instance, suppose that we have a class `language`, which contains objects such as `English`, `Spanish`, `Gaelic`, `French`, etc. The knowledge base might have a fact like

```
American[|nativeLanguage => language|].
```

which is read as a *type constraint* stating that, for every American, the property `nativeLanguage` can take only the values that are members of the class `language`. That is, in the above example, `Manuel[nativeLanguage->Spanish]` (or even `Gaelic`) would be ok, but `Manuel[nativeLanguage -> abracadabra]` is not. This is because, as we said earlier, `Spanish` and `Gaelic` are known to be in class `language`, but it is impossible to derive `abracadabra:language` given our knowledge base. Section 43.2 explains how type constraints can be checked and *ERGO* has also means of automatic enforcement of such constraints.

Like defaults, type information is inheritable. For instance, suppose the knowledge base has this information:

```
American[|nativeLanguage => language|].
PuertoRican::American.
Manuel:PuertoRican.
```

FLORA-2 would then derive the following by inheritance:

```
PuertoRican[|nativeLanguage => language|].
Manuel[nativeLanguage => language].
```

Note that since `Manuel` is a member of `PuertoRican` and not a subclass, it has lost the vertical bars during the inheritance, but the class `PuertoRican` still has them. From here on, nothing can be inheritable from `Manuel` even if this object can somehow be considered as a class. For instance, suppose `Manuel` has two alter egos: `ManuelJekyll:Manuel` and `ManuelHyde:Manuel`. Despite the fact that now `Manuel` has its own class members, since `Manuel[nativeLanguage->Spanish]` is a property of `Manuel` *as individual* and not as a class (the missing bars tell us that!), the property `nativeLanguage->Spanish` is not inherited to either `ManuelJekyll` or `ManuelHyde`. Similarly, `Manuel[nativeLanguage => language]` specifies the type of `Manuel` as an individual object and so it is inherited to neither of the two alter egos.

To illustrate the no-overriding property of signature inheritance, suppose the knowledge base also has `PuertoRican[|nativeLanguage => latinLanguage|]`. Unlike the inheritance of default values, this will not override the inheritance of the type `PuertoRican[|nativeLanguage`

`=> language|]`: it will still be inherited and this simply means that the actual values must lie in the intersection of the classes `language` and `latinLanguage`, which in this case is simply `latinLanguage`. However, this also means that one must be careful here. If the knowledge base had `American[|nativeLanguage => germanicLanguage|]` then `PuertoRican[|nativeLanguage => germanicLanguage|]` would also be true, but `Spanish` would not be in the intersection of the types `germanicLanguage` and `latinLanguage` (assuming our knowledge base is a good model of the real world). Such a mistake can be detected via type-checking (Section 43.2), which should prompt a revision of the way the types are specified. We should note, however, that an effect similar to inheritance overriding *can* be achieved via the mechanism of *defeasible reasoning* described in Section 38.

22.2 Monotonic Behavioral Inheritance

The default for behavioral inheritance is *non*-monotonic, so to request *monotonic* inheritance one must use the following compiler directive:

```
:- setsemantics{inheritance=monotonic}.
```

This semantic can also be requested at runtime by executing the command

```
?- setsemantics{inheritance=monotonic}.
```

(Of course, `?-` is to be used only when such a command appears in a file; it should be omitted on the $\mathcal{F}$LORA-*2* shell command line.) Here is an example of how monotonic behavioral inheritance works:

```
:- setsemantics{inheritance=monotonic}.
d(?_x):a(?_x).
g(?_x)::a(?_x).
a(r)[|b(y)->c|].
g(r)[|b(y)->e|].
a(u)[|d(1)|].
g(u)[|d(2)|].

?- d(?I)[b(?X)->?Y].

?I = r           // inherited information
?X = y
?Y = c
```

```
?- g(?I)[|b(?X)->?Y|].

?I = r           // inherited information
?X = y
?Y = c

?I = r           // explicitly specified information
?X = y
?Y = e

?- d(?I)[d(?X)].

?I = u           // inherited information
?X = 1

?- g(?I)[|d(?X)|].

?I = u           // inherited information
?X = 1

?I = u           // explicitly specified information
?X = 2
```

22.3 Non-monotonic Behavioral Inheritance

Non-monotonic behavioral inheritance is the default, but sometimes one might also need or want to specify it explicitly—either for documentation or to override a differently specified inheritance at runtime. The compiler directive (that would appear in a file) is

```
:- setsemantics{inheritance=flogic}.
```

and the runtime command is

```
?- setsemantics{inheritance=flogic}.
```

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

```
elephant[|color=>color, color->gray|].
```

```
royal_elephant::elephant.
clyde:royal_elephant.
```

The first statement says that the `color` property of an elephant must be of type `color` and that the default value is gray. The rest of the statements say that royal elephants are elephants and that `clyde` is an individual elephant. The question is what is the color of `clyde`? The color of that elephant is not given explicitly, but 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 once again that when class information is inherited by class' individual members, the resulting formula becomes object-level rather than class-level (i.e., `...[...]` instead of `...[|...|]`). On the other hand, when this information is inherited by a subclass from its superclass, then the resulting formula is a class-level formula because it should be still applicable to the members of the subclass and to 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 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 knowledge representation 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 -> security|].
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: `quaker[|policy -> pacifist|]` and `republican[|policy -> security|]`. In $\mathcal{F}$LORA-*2*, such conflicts cause previously established inheritance to be withdrawn and both `policy->pacifist` and `policy->security` become false for object `Nixon`. This behavior can be altered by adding additional rules and facts. For instance, adding `Nixon[policy->security]` would take precedence and override the inherited information. More generally, one could introduce priority over superclasses, say with a predicate `hasPriority`, and then add the rule

```
?Obj[policy->?P] :-
    ?Obj:?Class, ?Class[|policy->?P|], \naf hasPriority(?AnotherClass,?Class).
```

If we also had `hasPriority(republican,pacifist)` then inheritance from the class `republican` would take precedence.

Behavioral inheritance in F-logic is discussed at length in [18, 20]. 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:

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

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

1. `a[m->c]` was derived based on the belief that attribute `m` is not defined for the object `a`. However, once inherited, 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 give up the earlier conclusion made by inheritance as well as the subsequent inference made by the rule.

2. We did derive `a[m->d]` as a result of inheritance, but that's OK — we should not really be looking back and undoing 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`, `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 [10]. This approach is based on a fixpoint computation of non-monotonic behavioral inheritance. However, this semantics is unsatisfactory in many respects, including because it is procedural in nature. $\mathcal{F}$LORA-*2* uses a different, more cautious semantics for inheritance, which favors the first interpretation above.

Details of this semantics are formally described in [18]. Under this semantics, `clyde` will still inherit color white, but in the other two examples `a[m->c]` is *not* concluded. 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 false.
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 case

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

inheritance does not take place (`a[m->f]` is false), 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, *FLORA-2* views this agreement as accidental: 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-overridden 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, assuming that `c` and `d` are classes that are incomparable with respect to `::`, the following has a multiple inheritance conflict

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

   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 to be in conflict because here it is assumed that the method names are `m(k)` and `m(u)`, which are distinct names. Finally,

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

   is likewise *not* a conflict because inheritance here comes from the same class `c`.

In the examples that we have seen so far, path expressions queried only object-level information. To query class-level information using path expressions, FLORA-2 uses the symbol "!". For instance,

```
royal_elephant!color
```

means: some `?X` such that `royal_elephant[|color->?X|]` is true. In our earlier example, `?X` would be bound to `white`.

22.4 Inheritance of Negative Information

Certain kind of negative information can also be inherited, but inheritance goes in the opposite direction: from class members and subclasses to superclasses. The following subsections discuss this issue.

22.4.1 Negative Monotonic Behavioral Inheritance

If monotonic behavioral inheritance is requested using the `setsemantics` primitive, negative data frames propagate from objects to their classes as class-level frames, *i.e.*, as frames of the form `[|...|]`. Similarly, class-level data frames are propagated from subclasses to superclasses. For example, given the following data

```
:- setsemantics{inheritance=monotonic}.
obj:c1.
c1::c2.
obj[\neg prop->val].        // equivalently:  \neg obj[prop->val]
c1[|\neg prop2->val2|].     // equivalently:  \neg c1[|prop2->val2|]
obj[\neg boolprop].         // equivalently:  \neg obj[boolprop]
c1[|\neg boolprop2|].       // equivalently:  \neg c1[|boolprop2|]
```

the following queries will return the answers as shown:

```
?- c2[|\neg ?prop->?val|].  // equivalently:  \neg c2[|?prop->?val|]
?prop = prop
?val = val

?prop = prop2
?val = val2

?- c2[|\neg ?boolprop|].    // equivalently:  \neg c2[|?boolprop|]
```

```
?boolprop = boolprop

?boolprop = boolprop2
```

22.4.2 Negative Structural Inheritance

Since structural inheritance is monotonic, negative structural inheritance works similarly to negative *monotonic* behavioral inheritance. This means that negative signatures from class members propagate to become negative class-level (i.e., `[|...|]`) signatures for superclasses. Similarly, class-level signatures propagate from subclasses to superclasses. For instance, given

```
obj:c1.
c1::c2.
obj[\neg prop => type].       // equivalently:  \neg obj[|prop=>type|]
obj[\neg =>boolprop].         // equivalently:  \neg obj[|=>boolprop|]
c1[|\neg prop2 => type2|].
c1[|\neg =>boolprop2|].
```

the following queries show how the negative properties propagate towards the class `c2`:

```
?- c2[|\neg ?prop=>?type|].   // equivalently:  \neg c2[|?prop=>?type|]

?prop = prop
?type = type

?prop = prop2
?type = type2

?- c2[|\neg =>?prop|].       // equivalently:  \neg c2[|=>?prop|]

?prop = boolprop

?prop = boolprop2
```

22.4.3 Negative Non-monotonic Behavioral Non-Inheritance

For non-monotonic inheritance, propagation of negative information does not occur. First, it is a logical fallacy to expect to inherit negative information like this:

```
c[|\neg attr->1|].
obj:c.
cc::c.
```

inferring

```
?- obj[\neg attr->1].
?- cc[|\neg attr->1|].
```

This does not make logical sense because the fact `c[|\neg att->1|]` says that *it is known that* `att->1` *is not a default for class* `c`. One cannot logically conclude from this that it is also known that `att->1` is false for `obj` or that `att->1` is not a default for the subclass `cc` of `c`.

Neither does negative information propagate upwards the class hierarchy as was the case with monotonic inheritance of signatures and behavior. For instance, given

```
obj::c.
obj[\neg attr->1].
```

it does not follow that the default for `attr` in class `c` is not `attr->1` (i.e., `c[|\neg attr->1|]` has no logical justification; in fact, `c[|attr->1|]` could well be true).

The overall intuition for the desire to inherit negative information can nevertheless be achieved through a more powerful feature of *defeasible* reasoning described in Section 38. For instance, one could write a rule like

```
@{default(cl)} ?Obj[\neg att->1] :- ?Obj:cl.
```

which says that, by default, `obj[att->1]` is known to be false if `obj` happens to be a member of class `cl`. Other statements for the members of class `cl` may have higher priority and override the above. For instance,

```
abc:cl.
@{highpriority} abc[att->1].
\overrides(highpriority,default(cl)).
```

This says that, for the specific member `abc` of class `cl`, `att->1` is actually true because this information is specified with higher priority (`highpriority`) than the above default rule (`default(cl)`), as indicated by the priority fact `\overrides(highpriority,default(cl))`.

Despite all of the above, negative information does play a role in *blocking* inheritance. More specifically, explicit negative information specified for subclasses and class members is treated as explicit local statements that override (block) the inherited information. For instance, in

```
c[|attr1->{1,2}, attr2->{3,4}, attr3->{5,6}|].
cc::c.
obj:cc.
cc[|\neg attr1->1|].   // blocks inheritance of attr1->1 only
obj[\neg attr2->{}].   // blocks inheritance of attr2->anything
?- obj[attr1->?X].     // ?X = 2. Inheritance of atr->1 is blocked
?- cc[|attr1->?X|].    // same
?- obj[attr2->?X].     // no answers: all inheritance is blocked
?- cc[|attr2->?X|].    // ?X = 3,4: inheritance is blocked at obj, below cc
?- obj[attr3->?X].     // ?X = 5,6: inheritance occurs, nothing is blocked
?- cc[|attr3->?Y|].    // same
```

the explicit negative statement `cc[|\neg attr1->1|]` blocks the inheritance of `c[|attr1->1|]` down to both `cc` and `obj`. The explicit negative information `obj[\neg attr2->{}]` blocks the inheritance of `c[|attr2->val|]` down to `obj`, for *any* `val`. For `attr3` (and for `attr2` at the level of `cc`), however, no explicit blocking negative information exists, so the last three queries will report that data is inherited in full.

22.5 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[|foo(?Y) -> ?Z|] :- ?Z \is ?Y**2.
```

and every member of this class will then inherit exactly the same definition of `foo`. 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, 16.

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 it was studied in [19, 20]. 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 idea consists of three steps.

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

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

```
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 overridden by the code from `c1` and the latter (more specific code at `aa`) will be inherited by `aa`.

Step 1: define auxiliary methods.

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

Step 2: define the "fake" method `meth`.
Next we define `meth` — the method whose value inheritance will simulate the inheritance of the code of the methods `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 overridden by the more specific class `c1`.

Step 3: finishing up.
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 auxiliary 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 auxiliary 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 that of `bar/1` and if `bb` inherited the code of `foo/1`.

23 Custom Module Semantics

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

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

The following options are allowed:

Equality: `equality=none`, `equality=basic`, where `equality=none` is the default.

Inheritance: `inheritance=none`, `inheritance=flogic` `inheritance=monotonic`, where `inheritance=flogic` is the default.

Tabling: `tabling=reactive` (default), `tabling=passive`, `tabling=variant` (default), and `tabling=subsumptive`.

Custom: `custom=none`, `custom=`*`filename`*, where `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.

The following options are available only with the executable version of `setsemantics`, while the previously mentioned options can be used both at compile and run time.

Subclassing: `subclassing=strict`, `subclassing=nonstrict`, with *strict* being the default.

Class expressions: `class_expressions=on`, `class_expressions=none`; default: *none*.

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 a frame. The only safe way to execute `setsemantics` is in a query at the top level. For instance,

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

23.1 Equality Maintenance

User-defined equality. $\mathcal{F}$LORA-*2* users can define equality *explicitly* using the predicate `:=:`. For instance,

```
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 facts or in rule heads. 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 down the execution quite significantly. In order to be able to eat the cake and have it at the same time, $\mathcal{F}$LORA-*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 `:=:` is just a symmetric transitive relation that includes the identity. However, the 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) congruence.

If a $\mathcal{F}$LORA-*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 knowledge base does have such facts, then the default equality maintenance level is `basic`, because it is assumed that the use of `:=:` in the source is not accidental. In any case, the explicit `equality=...` option overrides the default.

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

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

will succeed, but the query

```
?- (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 knowledge engineer 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 27.)

Using the preprocessor to avoid the need for equality maintenance. One final bit of advice regarding equality: In many cases, knowledge engineers tend to use equality as an aliasing technique for long messages, numbers, etc. In this case, we recommend the use of 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 Web sites and that it fetches the corresponding Web pages, the above will fetch the page at the Yahoo site, because the FLORA-2 compiler will replace YAHOO with the corresponding string that represents a URL.

Limitations of equality maintenance in $\mathcal{F}$LORA-2. The implementation of equality in $\mathcal{F}$LORA-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 $\mathcal{F}$LORA-2, however, the query

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

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

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

will succeed.

23.2 Choosing a Semantics for Inheritance

As mentioned earlier, the `setsemantics` directive accepts three options: `inheritance=none`, `inheritance=flogic`, and `inheritance=monotonic`. The default is `flogic`; this type of inheritance is described in Section 22.

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.

Monotonic inheritance `inheritance=monotonic` is also sometimes appropriate—mostly in situations when information from superclasses is to be propagated and accumulated by subclasses and members without overriding. This type of inheritance is also significantly cheaper resource-wise than the F-logic inheritance, although it is more expensive than turning inheritance off completely.

23.3 Choosing a Semantics for the Subclass Relationship

The default semantics for the subclass relationship `::` in $\mathcal{F}$LORA-2 is *strict*. This means that there can be no loops in the subclass hierarchy. If $\mathcal{F}$LORA-2 detects a cycle at run time, it issues an

error. The user can change this semantics by executing the runtime directive

```
?- setsemantics{subclassing=nonstrict}.
```

Note that the above sets the nonstrict semantics for subclassing in the current module only. To change back to the strict semantics, one can execute

```
?- setsemantics{subclassing=strict}.
```

and to change the semantics in a different module one can execute

```
?- setsemantics{subclassing=nonstrict}@foo.
```

One can gind out the subclassing semantics in effect in the current module by executing the following query:

```
?- semantics{subclassing=?Sem}.
```

Note: the `subclassing` option is not available as a static `setsemantics` directive (i.e., it works only with " ?– " and not with ":–").

23.4 Choosing a Semantics for Tabling

The semantics for tabling can be specified along two different dimensions: *reactivity* and *identification*. The options for reactivity are `tabling=reactive` and `tabling=passive`, which is explained in Section 27.3. The options for the identification dimension are `tabling=variant` (the default) and `tabling=subsumptive`. Subsumptive tabling currently works only in conjunction with passive tabling, and it is experimental at present. In some cases, subsumptive tabling may significantly speed up computation time and save memory.

23.5 Class Expressions

FLORA-*2* defines a number of set-theoretic operations on classes. For instance, (a, b) is the intersection class, (a; b) is the union class, and (a − b) is the difference class. More precisely, (a, b) is the *maximal* subclass of a and b in the class hierarchy, and its extension is precisely the intersection of the extensions of a and b. The class (a; b) is the smallest superclass of a and b. The class (a − b) is the maximal subclass of a with extension that contains all the elements of a that are not in the extension of b.

The class expressions feature is *off* by default and must be enabled at runtime with

```
?- setsemantics{class_expressions=on}.
?- setsemantics{class_expressions=on}@somemodule.
```

It can also be disabled via

```
?- setsemantics{class_expressions=none}.
?- setsemantics{class_expressions=none}@somemodule.
```

Suppose the following information is given:

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

Then (`class1` − `class2`); `class3` has the extension of `a`, `b`, `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)|].
```

In the first case, the type is specified for `c1` as an individual object only. The second statement is about `c1` as a *class*, so this type specification is inherited by each member of the class `c1` and each subclass.

$\mathcal{F}$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 greatest lower 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 lowest upper 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 subclass 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 may adversely affect the performance of user knowledge bases, and FLORA-2 provides an optimization option that allows the user to disable these relationships for situations that do not need them, which is why this feature is off, by default.

Note: Type expressions introduce a potential for infinite answers for seemingly innocuous queries and so this feature is disabled by default, as explained earlier in this section. 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 FLORA-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 ":". □

23.6 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`.[14] To take advantage of this feature, the user must write the axioms using the same API that is used for FLORA-2 trailers, which are located in the `closure` directory of the distribution. This API will be described at a later date.

23.7 Querying the Module Semantics

In addition to the ability to change the semantics of a module, FLORA-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.
```

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

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

```
?- 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 determine which modules have a given combination of semantic options by examining every loaded module via the `isloaded{...}` primitive and then posing the desired `semantics{...}` query.

24 FLORA-2 and Tabling

24.1 Tabling in a Nutshell

Tabling is a fundamental deduction technique that augments query evaluation with a mechanism that remembers previously inferred conclusions. The result is a very efficient deductive engine.

FLORA-2 automatically tables frames and HiLog predicates, but allows the user to have non-tabled predicates as well. Such predicates are called *transactional* and are mostly used for side-effects, such as writing to or reading from files (or to/from the screen) and to insert and delete facts and rules. To indicate that a HiLog literal is non-tabled, it must be preceded 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. However, FLORA-2 issues a warning in case of such a double-use.

Symbols that are prefixed with the "`%`" sign can appear only as predicate formulas, predicate names, Boolean method names, or variables. A variable prefixed with "`%`" cannot 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 transactional Boolean methods
?- %?X@?M ~ %p(a).         // %p - a non-tabled predicate
```

but the following are not:

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

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

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

These rules also apply to queries issued against rule bases using the `clause{...}` primitive (described in Section 29) 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
```

24.2 Transactional Methods

Queries can have unintended effects when used in conjunction with predicates that have non-logical "side effects" (e.g., writing or reading to/from a file or a console) and queries that change the state of the underlying knowledge base. If a tabled construct (a HiLog predicate or an F-logic frame) has a side effect, the first time the predicate is called the side effect is performed. However, the *second* time the call simply returns success or failure depending on the outcome of the first call, since the answer is simply looked up in a table of previous answers. Indeed, tabled constructs represent purely logical statements that are not supposed to have side effects, so there is no reason to re-execute them. Thus, if a FLORA-2 construct is intended to perform a 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 *transactional*. Because by default FLORA-2 tables everything that looks like a frame, these transactional methods are potentially subject to the aforesaid problem.

To sidestep this issue, FLORA-2 introduces a new syntax to identify transactional methods — by allowing the "`%`" sign in front of a transactional 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, transactional methods can take arguments, but do not return any values. The only difference is that transactional methods are *not* tabled, while Boolean methods are.

Transactional signatures. Transactional methods can have signatures like other kinds of methods, which are specified as follows:

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

FLORA-2 does not support transactional methods specified as defaults at the class level. However, as seen from the second statement above, class-level signatures for transactional methods are supported.

24.3 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 *operational semantics* of FLORA-2. This operational semantics is essentially the same as in XSB and when no tabled predicates or frames are involved, the behavior is the same as in Prolog. However, when tabled HiLog predicates or frames (other than transactional methods) are used, the knowledge engineer must have some understanding of the way XSB evaluates tabled predicates.

Unlike Prolog, which computes answers one-at-a-time, FLORA-2 computes answers to the entire clique of inter-dependent predicates before the computation proceeds to the next subgoal in a rule body. The following little example 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 a frame and so it is implemented internally as a tabled XSB predicate. The second query is implemented as a non-tabled predicate. Thus, despite the fact that the two queries are *logically equivalent*, they are not *operationally equivalent*. 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 statements). 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 frame would be represented internally as a non-tabled predicate. Thus, the two queries, below,

```
?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.

24.4 Tabling and Performance

It is important to keep in mind that Prolog does not reorder frames and predicates during joins. Instead, all joins are performed left-to-right. Thus, rules and queries 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. XSB has partial support for a novel technique called *subsumptive tabling*, and it is known

that subsumptive tabling can speed up certain queries by an order of magnitude. However, XSB's implementation of subsumptive tabling does not support active tabling, which prevents most of the uses of this kind of tabling in FLORA-2.

24.5 Cuts

No discussion of a logic programming language is complete without a few words about the infamous Prolog cut (!). Although Prolog's 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 knowledge bases or implementing a pattern matching algorithm. To facilitate this kind of tasks, FLORA-2 lets the user use Prolog-like cuts.

A Prolog cut is akin to the switch/case-statements in languages like C or Java, but the full explanation of cuts is quite involved and we leave it out, assuming the reader is familiar with this concept.

Cuts across tables. The current implementation of XSB has a limitation that Prolog cuts cannot "cut tabled predicates." If you get an error message saying something about cutting across 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 frame or tabled HiLog predicate. However, it is OK to put a cut before any frame. It is even OK to have a cut in the body of a rule that *defines* a frame (again, provided that the body has no frame to the left of that cut). If you need to use cuts, plan on using transactional methods or non-tabled predicates.

The Prolog cut operator (!) in rule bodies may cause problems because XSB does not allow the cut to appear in the middle of a computation of a tabled predicate. For instance,

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

will not cause problems, 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 frame `?X[%foo(?Y)]` is implemented using 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 interdependent 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 tabled predicate `?X[foo->?Y]` and an error will result.

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

Cuts and facts. Prolog programmers are accustomed to treat facts as rules, and so Prolog programs with cuts heavily rely on the order of the rules. For instance, in

```
p(X,Y) :- q(X), !, r(X,Y).
p(3,4).
q(1).
r(1,2).
```

it is expected that the open query `p(X,Y)` will succeed from the first rule and will return `p(1,2)`. The query will not match the fact `p(3,4)` due to the cut because that fact comes after the first rule.

This kind of reasoning will not work in $\mathcal{F}$LORA-*2* because facts are stored in an internal database in an unordered fashion and one cannot know when they are going to be matched against the query. So, in $\mathcal{F}$LORA-*2* an open call `p(?X,?Y)` would return goth `p(1,2)` and `p(3,4)`. A user who wants to rely on rule ordering should convert the relevant facts into rules, as follows:

```
p(?X,?Y) :- q(?X), !, r(?X,?Y).
p(3,4) :- \true.
q(1).
r(1,2).
```

In that case, `p(?X,?Y)` will return only one answer.

25 User Defined Functions

User-defined functions (abbr., *UDF*) are a syntactic extension that permits the users to enjoy certain aspects of functional programming in the framework of a logic-based language.

25.1 Syntax

To define a UDF in $\mathcal{F}$LORA-*2*, one uses the following syntax:

```
\udf foo(t1,...,tn) := Expr \if Body.
```

`Expr` is a term and `Body` can be any formula that can be used as a rule body. The "`\if Body`" part is optional. The arguments of the UDF `foo` are terms, which usually are distinct variables.

`Expr` and *Body* can contain occurrences of other UDFs, but those UDFs must be defined previously. At first, this suggests that mutually-recursive UDFs cannot be defined. However, this is not the case, as will be explained shortly.

Instead of "if", *FLORA-2* also allows the use of the rule connective `:-`:

```
\udf foo(t1,...,tn) := Expr :- Body.
```

Example. The following simple example defines `father/1` as a function.

```
\udf father(?x):=?y \if father(?x,?y).
father(mary,tom).
father(john,sam).
```

So, instead of writing `father(John,?y)` and then using `?y` one can simply write `father(?John)`:

```
?- ?y=father(?x).
```

will return

```
?x=mary
?y=tom

?x=john
?y=sam
```

The query

```
?- writeln(father(mary))@\io.
```

will output

```
tom
```

Mutual recursion in UDF definitions. It was mentioned that any UDF that occur in a definition of another UDF must be defined previously. However, this does not mean that mutually recursive UDFs cannot be defined: one should just exercise care.

First, we should make it clear that recursive UDFs by themselves do not pose problems and can be used freely. For example, in

```
\udf foo(...) := Expr \if Body.
```

A more complex situation arises when we have mutual recursion, as in

```
\udf foo(...) := Expr-cannot-contain-bar \if Body-contains-bar.
\udf bar(...) := Expr-can-contain-foo \if Body-might-contain-foo.
```

This is not allowed, because `bar` is used in `Body-contains-bar` before it was defined. But there is a simple workaround: we can introduce a temporary predicate as follows:

```
\udf foo(...) := Expr-cannot-contain-bar \if tempfoo(...).
\udf bar(...) := Expr-can-contain-foo \if Body-can-contain-foo.
tempfoo(...) :- Body-contains-bar.
```

In this latter case, all uses of UDFs are defined previously, and UDFs are still mutually recursive. If `foo` has more than one `\udf`-clause and one of the clauses does not involve `bar` then one does not even need to introduce temporary predicates. For instance, the following is legal:

```
\udf foo(...) := Expr-contains-no-bar \if Body-contains-no-bar.
\udf bar(...) := Expr-can-contain-foo \if Body-can-contain-foo.
\udf foo(...) := Expr-can-contain-bar \if Body-can-contain-bar.
```

Here the first clause declares `foo` as a UDF, so the compiler is able to handle `foo` correctly in the second clause despite the fact that `foo` has not been fully defined yet at that point.

UDFs and modules. A UDF definition is a purely compile-time directive. It has an effect from the moment the function is defined till the end of the file (which extends to all the files included using the `#include` preprocessor directive). This implies that UDFs that occur inside calls to other *F*LORA-*2* modules are subject to the directives of the module where the functions occur, not in the module that is being called. For instance, consider

```
\udf func1(?x) := ?y \if pred1(?x,?y).
p(?x) :- q(func1(?x))@bar.
pred1(c,a).
```

Here `func1` will be compiled according to the function definition shown above. If the module `bar` has another definition for `func1`, that definition has no effect on the above occurrence of `func1(?x)`. Thus, the query

```
?- p(c).
```

results in a call to `q(func1(c))@bar`, and since `func1(c)` is defined to be `a` this is tantamount to executing the query

```
?- q(a).
```

Note that, since function definitions can contain anything that constitutes a rule body, calls to other modules are allowed. For instance,

```
\udf foo(?x):=?y \if pred(?x,?y)@bar, pred(?x,?y,?z)@moo, q(?z).
```

UDFs and updates. At present, only non-transactional UDFs are supported. If a UDF definition contains an update then a runtime error will result. For instance, if a UDF `foo(...)` has such a definition then the following error will likely occur at runtime:

```
++Error[FLORA-2]> non-transactional head-literal foo(...) := ?A depends on an update
```

Advanced examples. One of the most interesting examples of the use of UDFs arises when we define arithmetic operations as functions. For instance, normally one would write

```
?- ?x \is 1+2, writeln(?x)@\io.
```

but with UDFs we can define "+" as a function:

```
\udf ?x+?y := ?z }\if ?z \is ?x+?y.
```

and then issue the following query:

```
?- writeln(1+2)@\io.
```

The following example shows how we can define the Fibonacci function:

```
\udf ?x+?y := ?z \if ?z \is ?x+?y.
\udf ?x-?y := ?z \if ?z \is ?x-?y.
\udf fib(0) := 0.
\udf fib(1) := 1.
\udf fib(?x) := fib(?x-1)+fib(?x-2) \if ?x>1.
```

We can now ask the query

```
?- writeln(fib(34))@\io.
```

Note that the above example also illustrates that a definition of a UDF can consist of multiple `\udf`-statements—just like a definition of a predicate can have multiple rules.

Querying UDFs. $\mathcal{F}$LORA-*2* maintains meta-information about UDFs and they can be queried at runtime using the primitives `clause{...}` and `@!{...}`. For instance,

```
\udf foo(?X,?Y) := p(?Z) \if bar(?X,?Y,?Z).

flora2 ?-  clause{@!{?X[type->?T]} ?H,?B}.

?X = 4      // 4 happens to be the Id of that UDF
?T = udf
?H = foo(?_h1642,?_h1643) := _h1637
?B = (${bar(?_h1642,?_h1643,?_h1660)@main}, ?_h1637 = p(?_h1660))

flora2 ?-  clause{@!{?X[type->?T]} (?UDF(?,?) := ?s) ,?B}.

?X = 4
?T = udf
?UDF = foo
?B = ($bar(?_h5733,?_h5734,?_h5735)@main, ?_h5742 = p(?_h5735))

flora2 ?-  @!{?X[type->?T]}.

?X = 4
?T = udf
```

Note from the second query above that the head of a UDF can be queried using the (*udf* `:=` *result*) idiom. The parentheses around this idiom are important in that context. This is because the

expression `?UDF(?,?):= ?,?B` inside clause{...} above is parsed as `?UDF(?,?):= (?,?B)` and not as `(?UDF(?,?):= ?), ?B`.

25.2 Higher-order UDFs

UDFs can use arbitrary HiLog terms. These terms can be used both in the head of the UDF definition and in its body. For instance,

```
\udf abc(?x):=?x(a,b).
\udf ?x(?y,b)(?y):=t(?z) \if pred(?x,?y,?z).
\udf t(c):=d.
pred(foo,a,c).
q(d).
?- q(abc(?x)(?y)).
```

Perhaps it is not obvious, but the query succeeds with the answers `?x = foo`, `?y = a`. Indeed, `abc(?x)` rewrites to `?x(a,b)` by the first UDF definition, so the query becomes `q(?x(a,b)(?y))`. By the second (higher-order!) UDF, this rewrites to `q(t(c))` with the bindings `?x=foo` and `?y=a`. Finally, the query rewrites to `q(d)`, by the third UDF, and succeeds.

25.3 User-defined Functions and Multifile Modules

Recall from Section 17.5 that sets of rules can be added to an already loaded module using the `add`{*file*} or `[+`*file*`]` commands. If the already loaded module has UDF definitions, say, for `foo/2` and `bar/3`, and if the file `file.flr` uses those functions, then it is necessary to tell $\mathcal{F}$LORA-*2 in the file* `file.flr` that these are UDFs and not regular HiLog function symbols. If this is not done, the things are unlikely to work correctly and it would be very hard to find out why. To prevent this type of hard-to-find mistakes, the system will issue an error at loading time.[15]

To deal with such situations, $\mathcal{F}$LORA-*2* provides the `useudf` directive. For instance, if one puts the following at the top of `file.flr`

```
:- useudf{foo/2, bar/3}.
```

then `foo/2` (any occurrence of `foo` as a functor with two arguments) and `bar/3` (any occurrence of `bar` as a functor with three arguments) will be compiled correctly in `file.flr` and no errors will be issued.

[15] It is impossible to catch this mistake at compile time, since one cannot know in advance that `file.flr` is intended to be loaded into a module with existing UDF definitions. The `useudf` directive is used to supply precisely this kind of information to the compiler.

Note: UDF definitions and the `useudf` directives are intended to reside in *different* files (for the same UDF): it makes *no* good sense to have both a function definition and a `useudf` directive for the *same* UDF in the *same* file. However, $\mathcal{F}$LORA-*2* is tolerant to this kind of misspecification. □

To simplify the use of UDFs, $\mathcal{F}$LORA-*2* supports *implicit* `useudf` directives in certain cases. Namely, if a file with a UDF definition is loaded into a module then any file that will be *compiled* and then *added* to that same module in that $\mathcal{F}$LORA-*2* session will inherit the existing function definitions. In this case, there is no need to specify the `useudf` directive explicitly in `foo.flr`. However, if the file being added, say `foo.flr`, is compiled in a different $\mathcal{F}$LORA-*2* session (one that does not have the appropriate UDF definitions) then an explicit `useudf` directive would be required. Otherwise, a runtime error will be issued the next time `foo.flr` is added if there is a mismatch between the UDF definitions in these two different $\mathcal{F}$LORA-*2* sessions.

25.4 Semantics

Semantically, the functional notation is just a syntactic sugar. A function definition of the form

```
\udf foo(t1,...,tn) := Expr if Body
```

is converted into the rule

```
newpred_foo(t1,...,tn,Expr) :- Body.
```

Then every occurrence of the function in a rule head

```
head(...,foo(s1,...,sn),...) :- rulebody.
```

is rewritten into

```
head(...,?newvar,...) :- newpred_foo(s1,...,sn,?newvar), rulebody.
```

In the rule body, the occurrences of `foo(s1,...,sn)` are rewritten as follows:

```
... :- ..., pred(...,foo(s1,...,sn),...), ...
```

is changed to

```
... :- ..., newpred_foo(s1,...,sn,?newvar), pred(...,?newvar,...), ...
```

It is important to keep in mind that UDFs are *partial* functions and that any argument bindings for which an UDF is undefined falsifies the containing predicate or frame. For instance,

```
p(?).
\udf f(?x) := ?y \if cond(?x,?y).
cond(1,2).
?- p(f(1)).
Yes
?- p(f(2)).
No
```

Here `f(?x)` is a partial function that is defined only for `?x=1`. Since `f(1)=2` and `p(...)` is true of everything, `p(f(1))` evaluates to true. However, `p(f(2))` evaluates to false because `f(2)` is undefined and this falsifies `p(f(2))`. The rationale is that `p(...)` is true of every *defined* argument, but is false when an argument cannot be evaluated to a bona fide term.

25.5 Representing equality with user defined functions

An important application of UDFs is simulation of certain cases of equality. Simulating equality using UDFs is *orders of magnitude more efficient*. The equality that can be simulated in this way must be such that all terms that one wants to equate must be explicitly given at compile time. For instance, a term or its main function symbol cannot be represented by a variable (e.g., `?X` or `?X(?y,?z)`) and determined only at run time. Also, if the system of UDF definitions is not *confluent* (confluence is also known as the Church-Rosser property) then the fundamental property of substitutivity of equals by equals cannot be guaranteed.

For instance, equality of the following kind ?`x` = `abc`, ?`x`(`pqr`) :=: `cde` cannot be simulated because the term `abc(pqr)` that is equated to `cde` here is not explicitly given at compile time.

It is important to also keep in mind that UDFs provide only **unidirectional** rewriting. For instance, in

```
\udf a := b.
\udf a := c.
q(c).
?- q(b).
```

the query fails, since `b` is **not** rewritten back into `a` (and then into `c`).

To illustrate the cases where simulation is possible, suppose we want to state that any occurrence of `foo(?x)` should be equated to (rewritten into) `bar(?x)` if `pred(?x)` is true and `?x>2`:

```
\udf foo(?x):=bar(?x) if  pred(?x), ?x>2.
pred(1).
```

```
pred(2).
pred(3).
pred(5).
```

If we ask query

```
?- foo(?x)=bar(?x)
```

The answers will be

```
?x=2
?x=3
```

Here are some very simple, but useful examples of equality: rather than writing things like

```
foo(1) :=: 3.
foo(4) :=: 7.
```

we can instead define

```
\udf foo(1):=3.
\udf foo(4):=7.
```

The latter does not involve complex equality maintenance and is much more efficient.

The following example illustrates the issue of non-confluence of the equality through UDFs. Consider the following definitions:

```
\udf a := b.
\udf c := d.
\udf a := c.
```

and suppose `p(d)` is true. Then we can derive that `p(a)` and `p(c)` is true, but not that `p(b)` is true.

Note that if instead of `p(d)` we had `p(a)` then all these would be true: `p(b)`, `p(c)`, `p(d)`. This shows that one can go a long way with simulation of equality through UDFs, but care must be exercised. For example, if the set of UDF definitions is not confluent then substitution of equals by equals is not always guaranteed.

26 Controlling Context of Symbols

Using the same symbol in different contexts in the same knowledge base can be useful, but often can be the result of a typo, and such errors is very hard to catch. Examples of different contexts include the use of the same symbol as a function symbol and as a predicate, as a regular function symbol and a UDF, and even the uses of the same symbol with different arities.

FLORA-2 checks if the same symbol appears in different contexts and issues warnings if it finds such uses. For instance, in

```
p(?x,?y,?z):-q(?x,?y,?z).
p(?x):-t(?x).
```

the predicate `p` is used with arity 3 and 1, which may or may not be a mistake. In the following,

```
p(?x):-q(?x).
f(p(?x,?y)):-t(?x).
```

`p` occurs both as a predicate and a function symbol, and with different arities on top. In the next example we encounter the symbol `f` in the role of a UDF and also as a predicate:

```
\udf f(?x):=?y if ff(?x,?y).
t(?x):-f(?x).
```

and, in the next example, `tp` has both transactional and non-transactional occurrences.

```
%tp(?x,?y):-s(?x,?y).
tp(?x,?y,?z):-q(?x,?y,?z).
```

UDFs are also not allowed to have the same name as *FLORA-2* builtins, such as `isatom...`, `isground...` and the like.

In all these cases, warnings will be issued. However, the user can turn off the warnings that she believes not to indicate an error. This is done with the help of the directive `symbol_context`. The syntax is the following

`:- symbol_context{`*comma-separated-list-of-terms*`}`

The terms in the aforementioned list can be function symbols, predicates, or UDFs. For each type there is a separate idiom to indicate what is to be permitted. The legal specifications are as follows:

1. `termName(?,...,?)@?`, `termName(?,...,?)@\@`, `termName(?,...,?)@moduleName` — indicates that `termName` is a predicate of the given arity. The first form applies to all modules, the second applies to predicates that appear without any module specification (e.g., `abc(1,2)`), and the third applies to predicates in a specific module, `moduleName`.

2. `termName(?,...,?)` — indicates that `termName` is a term (not a predicate).

3. `udf termName(?,...,?)` — indicates that `termName` is a UDF.

4. `termName(*)@moduleName`, `termName(*)`, and `udf termName(*)` — these are wildcards that have the same meaning as above except that the warnings for the terms of the form `termName(...)` are suppressed for all arities in the corresponding contexts (predicate, HiLog term, UDF).

5. `termName/N`, where `N` is a non-negative integer — suppresses warnings for terms of the form $\texttt{termName(arg}_1\texttt{,...,arg}_N\texttt{)}$ in *any* context (predicate, HiLog term, or a UDF).

6. `termName/?` — like `termName/N` above but suppresses warnings for all arities. Thus, warnings will not be issued for `termName(...)` in all contexts and for all arities.

7. `warnoff` — suppresses symbol context warnings, but not errors.

8. `constoff` — suppresses warnings due to constants appearing in multiple contexts. This not only affects warnings, but also performance. In very large fact bases, the number of constants is typically very large. For each constant, $\mathcal{F}$LORA-*2* would normally create an entry in the `.fls` file, which is used to check for multiple contexts. For a large file, such an `.fls` file can be much larger than the original set of facts. With the `constoff` option, entries describing constants will not be saved in the `.fls` file, which can significantly improve the compilation and load time.

In the above, the arguments must be anonymous variables and `moduleName` can be either a module name or an anonymous variable. Examples:

```
:- symbol_context{p(?,?), q(?)@?, r(?,?)@bar, udf foo(?)}.
```

Once this directive is issued, the compiler will no longer warn about the occurrences of `p` as a term with two arguments, `q` as a predicate with one argument in *any* module, `r` as a binary predicate in module `bar`, and `foo` as a unary UDF.

Important: In the above, `termName` must be a constant, not a general HiLog term. For instance, `symbol_context{p(?,?)(?,?,?)}` is not allowed. That is, it is not possible to allow `p(?,?)` to appear as a functor in 3-ary HiLog terms while disallowing it in other contexts. In

other words, only `symbol_context{p(?,?)}` is supported, and this would allow `p(?,?)` to appear anywhere as a term (e.g., in `p(?,?)(?,?,?)`, in `p(?,?)(?,?)`, and in `p(?,?)(?,?,?)(?)`).

The `symbol_context` directive also has special syntax for sensors and predicates declared using the `:- prolog{...}` directive:

```
:- symbol_context{sensor foo(?,?), prlg bar(?,?,?)}.
```

Note that *FLORA-2* issues warnings only on finding a *second* inconsistent use of the same symbol. Since most symbols are used in just one context, there is no need to supply the above directive in most cases. For instance, in the following

```
:- symbol_context{p(?,?)}.
p(?x):-q(?x), t(?x).
f(p(?x,?y)):-t(?x).
```

one needs to notify the compiler only about the use of `p(?,?)` as a function symbol, since there is prior use of `p/1` as a predicate. The other symbols are used consistently and there is no need to do anything special for them.

27 Updating the Knowledge Base

FLORA-2 provides primitives to update the runtime database. Unlike Prolog, *FLORA-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 of an object contains *both* the facts that were *inserted explicitly* into the knowledge base and the facts that are specified implicitly, via rules. For instance, in

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

the fact `p(a)` will be placed in the base part of the predicate `p/1` 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.
```

FLORA-2 updates can be *non-transactional,* as in Prolog, or *transactional*, as in Transaction Logic [3, 2]. We first describe non-transactional updates.

27.1 Non-transactional (Non-logical) Updates

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

FLORA-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 frames. The optional part, `|`*query*, is an additional condition that must be satisfied in order for *literals* to be inserted. 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

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

the first statement inserts a particular frame. 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:

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

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

```
?- 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):

```
?- insert{Mary[children->?X]@foobar | adult(?X)@infomod}
```

Updates to Prolog modules is accomplished using the usual Prolog `assert/retract`:

```
?- assert(foo(a,b,c))@\prolog.
```

The following subtleties related to updates of Prolog modules are worth noting. Recall Section 19.4 on the issues concerning the difference between the HiLog representation of terms in *FLORA-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 avoid this problem, use explicit conversion:

```
?- 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, *FLORA-2* provides a higher-level primitive, `@\prologall` (equivalently `@\plgall`), as described in Section 17.8. This module specifier does automatic conversion of terms to and from Prolog representation, so the above example can be written much more simply:

```
?- 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, if `foo/3` was defined as a dynamic predicate in the module `bar`, we have to write:

```
?- 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

```
?- assert(foo(f(a)@\prolog(bar),b,c)@\prolog(bar))@\prolog.
```

or `@\prologall`:

```
?- 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 frames and predicates. The optional part, `|`*query*, represents an additional constraint or a restricted quantifier, similar to the one used in the insertion primitive.

For instance, the following predicate:

```
?- 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* $\wedge$ *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.

*F*LORA-*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 a 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:

   ```
   ?- 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.

27.2 Transactional (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 meaning of *literals* and *query* is the same as in Section 27.1.

The new update operators `t_insert`, `t_insertall`, `t_delete`, `t_deleteall`, `t_erase`, and `t_eraseall` work similarly to the non-transactional `insert`, `delete`, `deleteall`, `erase`, and `eraseall`, respectively, except that the new operations are *transactional*. Please refer to Section 27.1 for details of the non-transactional update operators.

The keywords `tinsert`, `tinsertall`, `tdelete`, `tdeleteall`, `terase`, and `teraseall` are also understood and are synonymous to the `t_*` versions of the transactional operators.

To illustrate the difference between transactional and non-transactional updates, consider the following execution trace immediately after the FLORA-*2* system starts:

```
flora2 ?-  insert{p(a)}, \false.

No

flora2 ?-  p(a).

Yes

flora2 ?-  t_insert{q(a)}, \false.   // or tinsert{q(a)},

No
```

```
flora2 ?-  q(a).

No
```

In the above example, when the first `\false` 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 `\false` 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, hence 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` go against the very concept of transactional updates.

- `fail`-loops are not going to work (will leave the database unchanged) for obvious reasons. The loop forms `\while` and `\until` should be used in such situations.
- Tabled predicates or methods must never depend on transactional updates. First, as explained on page 148, 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 knowledge engineer. 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 `t_*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 a cut or the `\one` directive.

27.3 Updates and Tabling

Changing predicates on which tabled predicates depend. By default, *FLORA-2* uses an advanced form of tabling, called *reactive* tabling. To understand what it is, consider the following knowledge base.

```
p(?X) :- q(?X).
q(1).
?- p(?X).
```

Loading this little example into FLORA-2 will yield the answer `?X = 1`, as expected. Next, suppose we add `q(2)` to the knowledge base and ask the same query, `p(?X)`. This time, the answer will be both `?X = 1` and `?X = 2` — also as expected. Although one tends to take this behavior for granted, it is useful to understand what is going on here.

After the first query, the query itself and its single answer is recorded in a table. All subsequent calls to that query are supposed to be answered just by a table lookup, without using the rules that define `p(?X)`. However, when `q(2)` is added, the system *reacts* and updates the answer table for our query. This is why we got the correct result when the query was issues the second time. This is achieved through the mechanism of *reactive tabling* of the underlying XSB inference engine. Natural as it is, implementing this type of tabling is very hard and XSB is the *only* logic programming system that supports it. Reactive tabling is similar to materialized view maintenance in commercial database systems. However, commercial DBMS have an easier job, as none of them does maintenance for recursive views. Even for non-recursive views, maintaining materialized views is not straightforward from the user point of view in such systems.

Passive (non-reactive) tabling mode. Although reactive tabling is what one usually needs, it is more expensive computationally. In some cases, a knowledge base might not be making any updates at all and paying the computational price of reactive tabling would be unjustified. For this case, FLORA-2 allows the user to request *passive tabling* for any module.

In FLORA-2 modules that use passive tabling, tables are not updated after they are constructed in response to queries. Therefore, subsequent updates followed by further queries in such modules may return incorrect (stale) answers. For instance, in the above example, the second query `p(?X)` will return a stale answer in which `?X = 2` will be missing.

To request passive tabling in a module at compile time, use this directive:

```
:- setsemantics{tabling=passive}.
```

Passive tabling can also be requested at run time, in which case the mode will change from reactive to passive on-the-fly:

```
?- setsemantics{tabling=passive}.
```

However, runtime changes to the tabling mode is not recommended and there are certain restrictions. For instance, this can be done only as a top-level query, not as part of any other query.

Interaction between passive and reactive tabling. In FLORA-2, passive modules can issue calls to reactive modules and vice versa. However, inter-module queries (between passive and

reactive and between passive and passive modules) will not be updated reactively in this case. Consider the following example:

```
// Module m1 (reactive, by default)
r(a).
s(?X) :- r(?X).
%upd :- insert{r(b)}.
```

```
// Module m2 (passive)
:- setsemantics{tabling=passive}.

p(?X) :- s(?X)@m1.
```

If the query `p(?X)@m2` is issued first, one gets the answer `?X=a`. Suppose next `%upd@m1` is invoked to update `r/1`. If then `p(?X)@m2` is asked again, the answer will still be `?X=a` even though `s(?X)@m1` would be reactively updated to include the answer `?X=b`. The reason is that `p(?X)@m2` is maintained passively, and it will not be alerted to the change in `s(?X)@m1`. Similarly, if `m1` were passive and `m2` reactive, the query `p(?X)@m2` will still not be updated, but for a different reason: here `s(?X)@m1` will not get updated, since in that case it would be maintained passively.

More generally, passively tabled queries invoked anywhere in the call-chain will stop update propagation up the chain. To illustrate, consider the following example:

```
// Module m1 (reactive, by default)
r(a).
s(?X) :- r(?X).
p(?X) :- q(?X)@m2.
%upd :- insert{r(b)}.
```

```
// Module m2 (passive)
:- setsemantics{tabling=passive}.

q(?X) :- s(?X)@m1.
```

Here module `m1` calls `m2` and the latter calls back. If the queries `s(?X)@m1` and `p(?X)@m1` are issued then one gets `?X=a` as an answer in both cases. Suppose now that `%upd@m1` is invoked thereby changing `r/1`. Since `m1` is reactive, the query `s(?X)@m1` will now return both `?X=a` and `?X=b`. On the other hand, `p(?X)@m1` will still return only the first answer because it calls `q(?X)@m2` and the change to `r/1` is not propagated to `q(?X)@m2`, since the latter is defined in a passive module.

Explicit refresh of passive tables. What if updates in certain modules are very rare, but queries are frequent and time is money? In this case, passive tabling might still make sense. $\mathcal{F}$LORA-*2* provides a partial redress in the form of the `refresh{...}` operator. This operator lets the knowledge engineer explicitly remove stale answers from tables. For instance, in the above example we could do the following:

```
flora2 ?-  refresh{p(?)}, p(?X).
```

and get the right result. In general, `refresh{...}` can take a comma-separated list of facts to be purged from the tables, and these 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}.
```

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 ?-  \abolishtables.
```

can be used. However, this command should be executed only as a *standalone* query. Also, neither `refresh{...}` nor `\abolishtables` can occur under the scope of the negation operator `\naf` (either directly or indirectly).

Note that both `refresh` and `\abolishtables` can be also used in modules that use reactive tabling. However, in this case, `refresh` has no effect and `\abolishtables` will affect only the passive modules, if there are any.

Tabled predicates that depend on update operations. A related issue is that a tabled predicate (or a frame literal) 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 transactional 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 transactional side-effect by default, this warning is also issued when a tabled literal depends on non-tabled HiLog predicates. $\mathcal{F}$LORA-*2* also imposes restrictions on the use of updates that affect tabled facts: it does not allow such dependency to occur in modules with reactive tabling. $\mathcal{ERGO}$ does allow such dependency by providing the special feature of "stealth mode" updates.

Here is an example of a situation where 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 and thus silence the warnings.

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 in this case. 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)}`.

We should also mention here that executing the command `warnings{compiler}` in the FLORA-*2* shell will turn the dependency checking off globally. In some cases, this can reduce the compilation time, but is discouraged except when debugging is finished.

27.4 Updates and Meta-programming

The update operators can take variables in place of literals to be inserted. For instance,

```
?- ?X ~ a[b->c], insert{?X}.
```

Here ~ is a meta-unification operator described in Section 19.1. 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.
```

27.5 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 frames that involve non-transactional updates does not have logical semantics, and this requires the knowledge engineer to have a certain understanding of the *operational* semantics of $\mathcal{F}$LORA-*2* (Section 24.3). In this case, the semantics of negation applied to methods or predicates that produce side-effects through updates or I/O is that of Prolog negation (Section 20.1).

When only transactional updates are used, the semantics is well defined and is provided by *Transaction Logic* [3, 2]. In particular, negation is also well defined. However, simply negating an update, A, is useless, since logically it means jumping to some random state that is not reachable via execution of A. As explained in [3, 2], negation is typically useful only in conjunction with $\wedge$, where it acts as a constraint, or with the hypothetical operator of possibility `<>`. In most cases, when the knowledge engineer wants to apply negation to a method that performs logical (transactional) updates, he has $\sim$`<>`*method* in mind, i.e., a test to verify that execution of *method* is not possible. Both $<>$ and $\sim$`<>` (equivalently `\naf<>`) are supported in $\mathcal{F}$LORA-*2*.

The idiom `\neg` %*method* is considered to be an error, but `\+` %*method* and `\naf` %*method* (the latter interpreted as `\+` %*method*) are permitted. However, they yield meaningful results only if %*method* has no side effects and is non-recursive.

A transaction of the form ?– `<>`*expression* is true in the current state if the transaction ?– *method* *can* be executed in the current state. The state, however, is not changed even if ?– *method* does change the state during its execution. In other words, ?– `<>`*expression* only *tests* if execution is possible. The transaction ?– $\sim$ `<>`*expression* tests if execution of *expression* is *im*possible. It is true if ?– *expression* fails and false otherwise. However, regardless of whether ?– *expression* fails or not, the current state underlying the knowledge base does not change. Note

that `<>` and sin`<>` can apply to a group of subgoals, if they are enclosed in parentheses. If the expression does not change the underlying state of the knowledge base then ?– `<>` *expression* and ?– ∼`<>` *expression* reduce to ?– *expression* and ?– `\naf` *expression*, respectively. Examples:

```
?- <> %p(?X).
?- <> (%p(?X), q(?X,?Y), %r(?Y)).
?- ~<> (%p(?X), q(?X,?Y), %r(?Y)).
?- ~<> (%p(?X), q(?X,?Y), <> %r(?Y)).
```

27.6 Counter Support

It is often necessary to maintain a global counter, which can be set, queried, and updated. In principle, this can be done by designating a certain unary predicate and updating its content as necessary. However, this is a bit cumbersome, and FLORA-*2* provides a more efficient way. A counter is just an FLORA-*2* symbol that can be accessed via the following operations:

- `counter{Name := Integer}` — set counter
- `counter{Name := ?Var}` — query counter
- `counter{Name + Integer}` — increment counter
- `counter{Name − Integer}` — decrement counter

For example,

```
?- counter{abc:=3}, counter{abc+5}, counter{abc=?X}.
?X = 8.
```

28 Insertion and Deletion of Rules

FLORA-*2* supports *non-transactional* insertion of rules into modules as well as deletion of inserted rules. A module in FLORA-*2* gets created when a file is loaded into it, as described in Section 4, 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 27, FLORA-*2* predicates and frames can have both static and dynamic parts and no special declaration is required

to make a predicate dynamic. The same frame 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.

28.1 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 knowledge base.

A dual operation to module creation is `erasemodule` with the following syntax:

`erasemodule`{*modulename*}

The primitive `erasemodule` empties module out but does not delete it from the registry of modules, so new facts and rules can be inserted into that module again without the need to recreate that module with another `newmodule` command.

28.2 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`. The reason for having three different insertion commands is the same as in Prolog: the position of a rule with respect to other rules may sometimes have an affect on performance or query termination.

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`, and *rulelist* is a comma-separated list of rules. The rules being inserted *should not* terminate with a period (unlike the static 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.

Note: rule Ids and other meta-data (see Sectin 36) can be supplied with the insert operator:

```
?- insertrule{@!{abc[foo->bar]} ?X:student :- %enroll(?X,?_T)}.
```

Meta information can also be supplied with the `deleterule` operator.

28.3 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 must be enclosed in parentheses and *should not* terminate with a period.

To delete the rules inserted in the second example of Section 28.2, we can use

```
?- deleterule_a{(?X:student :- %enroll(?X,?_T)),
                (?X:student :- %enroll(?X,?_T))@mod1}.
```

or

```
?- 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, the rule deletions below are all valid:

```
?- deleterule{(?H:-q(?X))@foo}.
?- deleterule{(p(?X):-q(?X))@?M}.
?- deleterule{?H:-?B}.
```

The last query attempts to 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

```
?- deleterule{o[b->?V1] :- something(?V1,?V2)}.
```

will succeed and will delete the first of the above rules. Therefore, the following action will subsequently fail:

```
?- deleterule{o[b->?V1,d->?V2] :- ?Body}.
```

The problem of Cuts What is behind rule insertion is pretty simple. As we know from Section 27, 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 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 cuts, the knowledge base might not behave as expected. For example, if we have the following rules:

```
p(?X) :- r(?X).
r(a).
q(b).
?- insertrule_a{p(?X) :- q(?X),!}.
?- p(?X).
```

we would 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`.

29 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 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:

```
?- insertrule_a{tc(?X,?Y) :- e(?X,?Y)}.
?- insertrule_a{tc(?X,?Y) :- tc(?X,?Z), e(?Z,?Y)}.
?- newmodule{foo}.
?- insertrule_a{(tc(?X,?Y) :- e(?X,?Y)@\@)@foo}.
?- insertrule_a{(tc(?X,?Y) :- tc(?X,?Z), e(?Z,?Y)@\@)@foo}.
```

Then the query

```
?- 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

```
?- clause{tc(?X,?Y)@foo,?Z}.
```

We can also query rules by providing patterns for their bodies. For example, the query

```
?- clause{?X, e(?_,?_)}.
```

will return the first and the third rules.

Querying rules with composite heads involves the following subtlety. Recall from Section 28.3 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,

```
?- 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:

```
?- 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.

`clause{`*type,head,body*`}`

For example,

```
?- clause{static,?X,?Y}.
?- clause{dynamic,?X,?Y}.
```

Within 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):

```
?- clause{dyna,?X,?Y}.
?- clause{dynz,?X,?Y}.
```

Due to a limitation of the underlying Prolog system, the `clause{...}` primitive cannot query rules whose size exceeds a 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.

The `clause{...}` statement can also be used to query by the *rule Id* and other meta-data (rule descriptors) associated with $\mathcal{F}$LORA-*2* rules. This form of the clause statement is described in Section 36.4. Here is an example of querying by rule Id:

```
?- clause{@!{myrule1} ?Head, ?Body}.
```

Note that $\mathcal{F}$LORA-*2* facts are different from rules. They cannot be queried using the `clause` primitive. Instead, the primitive `isbasefact{...}` should be used for that. For instance,

```
a:b.
?- isbasefact{a:b}.
Yes

?- clause{a:b,?}.
No
```

30 Aggregate Operations

Aggregate operators play important role in data analytics applications and are commonly used in database languages. This section describes the aggregate operators available in $\mathcal{F}$LORA-*2*.

30.1 Syntax of Aggregate Operators

An aggregate query in $\mathcal{F}$LORA-*2* has the following form:[16]

```
agg{?X | query}
agg{?X[?Gs] | query}
```

where

[16] The syntax for aggregates is similar to that used in the FLORID system http://www.informatik.uni-freiburg.de/~dbis/florid/.

- `agg` is the *aggregate operator*; it can be one of these: `min`, `max`, `count`, `countdistinct`, `sum`, `sumdistinct`, `avg`, `avgdistinct`, `setof`, `bagof`.
- `?X` is called the *aggregation variable* (it must be a variable, not any other term!).
- `?Gs` is a list of comma-separated *grouping variables*.
- *query* is a logical formula that specifies the query conditions; it has the form of a rule body formula (including conjunctions, disjunctions, quantifiers, and even nested aggregation).

All the variables appearing in *query* but not in `?X`, `?Gs`, and not appearing prior to the aggregation 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`.

The `setof` and `bagof` aggregates have three additional forms:

```
agg{?X(SortSpec) | query}
agg{?X[?Gs](SortSpec) | query}
agg{?X[?Gs](SortSpec1,SortSpec2) | query}
```

where `SortSpec` specifies how the result of the `setof`. The exact syntax for this is described in Section 30.4.

Here are a few typical uses of aggregation:

```
?- ?Z = min{?S|John[salary(?Year)->?S]}.
?- ?Z = setof{?Pair|John[salary(?Year)->?S], ?Pair=pair(?Year,?S)}.
?- ?Z = setof{?Pair([asc(2)])|John[salary(?Year)->?S], ?Pair=pair(?Year,?S)}.
```

The first example binds `?Z` to the smallest salary that John has ever had. The second example binds `?Z` to a list of year-salary pairs that represents the earning history of John. This illustrates how to overcome the syntactic restriction that the aggregation variable cannot be a general term. In other words, this restriction does not limit the expressive power. The third example is like the second, but the year-salary list will be sorted by the year in the ascending order (`asc(2)` refers to the first argument of `pair(...,...)`). More on sorting is found in Section 30.4.

30.2 Evaluation of Aggregates

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`, `countdistinct`, `sum`, `sumdistinct`, `avg`, `avgdistinct`, `setof`, and `bagof`.

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`, `avg`, `sumdistinct`, and `avgdistinct` can take numbers only. If the aggregate variable is instantiated to something other than a number, these operators will discard it and produce a runtime warning message.

The operator `bagof` collects all the bindings of the aggregation variable into a list. The operator `setof` works similarly to `bagof`, except that all the duplicates are removed from the result list and the resulting list is sorted lexicographically. Note that these operators create ordered lists, not sets, unlike what as the operator names might suggest.

Caveat: `setof` eliminates all duplicates only if the bindings for the aggregate variable are all ground HiLog or Prolog terms. If some terms are non-ground or are reified formulas, some duplicates may remain. To tell `setof` to eliminate them all, execute the query

```
?- setruntime{setof(strict)}.
```

The strict `setof` mode incurs certain overhead, especially if the number of elements in the list returned by `setof` is large. Therefore, it is a good practice to return to the default "lax" mode when possible:

```
?- setruntime{setof(lax)}.
```

See Section 43.8.8 for more details on this option.

Note: The aggregates `min`, `max`, `avg`, and `avgdistinct` fail if *query* produces no answers. In contrast, `sum`, `count`, `sumdistinct`, and `countdistinct` return 0 in such a case, and the operators `bagof` and `setof` return the empty list.

The difference between `sum`, `count`, and `avg` on the one hand and `sumdistinct`, `countdistinct`, and `avgdistinct` on the other is that the latter eliminate duplicates from the bindings produced by *query*. Thus, for example, if *query* binds the aggregate variable to, say 31, more than once then `countdistinct` will count this only once (and `sumdistinct`, `avgdistinct` will consider this binding only once also), while `count` (respectively, `sum` and `avg`) will consider 31 as many times as it was produced by the query. Example:

```
p({31,45}).
p(?X) :- ?X=31.
```

Here `count{?X|p(?X)}` yields 3 and `sum{?X|p(?X)}` evaluates to 107, while `countdistinct{?X|p(?X)}` yields 2 and `sumdistinct{?X|p(?X)}` produces 76. This is because the answer `p(31)` will be derived twice: once via one of the above facts and once via the rule. (One might not realize this by posing the top-level query `?- p(?X)`, since *FLORA-2* eliminates duplicate answers before showing them to the user).

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

```
?- ?Z = min{?S|John[salary(?Year)->?S]}.
?- ?Z = count{?Year|John.salary(?Year)<max{?S|John[salary(?Y)->?S], ?Y<?Year}}.
?- avg{?S[?Who]|?Who:employee[salary(?Year)->?S]} > 20000.
```

If an aggregate contains grouping variables then this aggregate would backtrack over such grouping variables. In other words, grouping variables are considered to be existentially quantified (and the scope of that quantifier is the entire rule body). 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 `setof` aggregate.

```
flora2 ?-  ?Z = setof{?Year[?Who]|?Who[salary(?Year)->?X], ?X < 60}.
?Z = [1990,1991]
?Who = Mary

?Z = [1990,1991,1997]
?Who = John
```

30.3 Scope for Variables in Aggregate Operators

Another very important aspect is that the aggregation variable, `?X`, has the scope that is restricted to *query*. The scope of the grouping variables, `?G`, is not limited to the query, but it is usually *meaningless* for them to occur *to the left* of that scope. (The semantics of the aggregates will be explained shortly.) Such an occurrence will cause a compile-time warning and, possibly, a run-time error. To illustrate, consider the following rules:

```
head :- p(?X), r(?X), ?P = sum{?X | q(?X)}.                  // rule 1
head :- p(?X,?W), r(?X), ?P = avg{?X[?V,?W] | q(?X,?V,?W)}.  // rule 2
head :- r(?X), ?P = min{?X[?V,?W] | q(?X,?V,?W)}, p(?X,?W).  // rule 3
```

In the first rule, the first two occurrences of `?X` are outside of the scope of `sum{...}` where `?X` is used as an aggregation variable. Therefore, the first two occurrences represent variables that are distinct from the variable used in the aggregation. This is similar to scoping of variables by universal and existential quantifiers. In the second rule, the first two occurrences of `?X` are likewise unrelated to the last two occurrences of `?X` in `avg{...}`. The occurrence of `?W` to the left of the aggregation is likely a logical error caused by a misunderstanding of grouping (or, at best, is meaningless). The user probably needed this instead:

```
head :- p(?X,?W), r(?X), ?P = avg{?X[?V] | q(?X,?V,?W)}.     // rule 2'
```

If the user thought `?W` was really needed among the grouping variables, then a different behavior must have been sought. FLORA-*2* is zealous about this and will issue an stern warning, if it finds such occurrences.[17] Moreover, if, in the second rule above, p/2 binds `?W` to a ground term then a runtime error will be issued because it makes little sense to keep a constant in a grouping list. It the user really understands what he is doing, he can avoid both the error and the warning by rewriting the second rule as

```
head :- p(?X,?WW),r(?X),?P = avg{?X[?V,?W] |q(?X,?V,?W),?W=?WW}. // rule 2''
```

but most likely the intent was rule 2' above where `?W` was simply removed from the grouping list.

In contrast, the third rule above is proper (since `?W` occurs to the right, not left, of the aggregation) but, perhaps, renaming the first `?X` would have been an improvement.

30.4 Aggregation and Sorting of Results

The aggregate functions `setof` and `bagof` also have other forms, which supports sorting of the output of these aggregates:

`agg{?X(`*SortSpec*`) | `*query*`}`
`agg{?X[?Gs](`*SimpleSortSpec*`) | `*query*`}`
`agg{?X[?Gs](`*SortSpec1,SortSpec2*`) | `*query*`}`

[17] In some cases (e.g., in a disjunction like `p(?X) ; ?Z = sum{?P[?X]|q(?P,?X)}`), FLORA-*2* might issue a false warning. The user can circumvent this by renaming one of the occurrences of `?X`.

where `agg` can be `setof` or `bagof`.

A *SortSpec* specifies how the output is to be sorted. It is either the constant `asc` (ascending), `desc` (descending), or a list of the form `[`*spec1*`,...,`*specN*`]`, where each component has the form `asc(`*N*`)` or `desc(`*N*`)` for some positive integer *N*. A *SimpleSortSpec* is either `asc` or `desc`.

The first form of the `setof/bagof` operator does not have grouping and the sort specification controls the order in which the values of `?X` that satisfy *query* will appear in the result of the aggregation. As the names suggest, `asc` means ascending lexicographic ordering and `desc` means that the order is descending. If the sort specification has the form `[`*spec1*`,...,`*specN*`]`, the control of the sorted order is finer. The elements of the aggregation result will be sorted first according to `spec1`, then `spec2`, etc.

Recall that each *spec-i* has the form `asc(`*N*`)` or `desc(`*N*`)`, where *N* refers to a component of each member of the aggregation result. In HiLog terms or reified HiLog predicates, 1 refers to the functor of the term (or the predicate name), 2 refers to the first argument, etc. In a reified frame `${obj[prop->val]}`, 1 refers to `obj`, 2 to `prop`, and 3 to `val`. In reified binary formulas, such as `${L:=:R}`, `${L:R}`, or `${L::R}`, 1 refers to `L` and 2 to `R`. For lists, 1 refers to the first list element, 2 to the second, etc. Note that if *N* refers to a non-existing component of a formula, an error will be issued from the Prolog level.

The second form of the above aggregates uses a simplified form of the sort specification (i.e., `asc` or `desc`). It provides a simple way of controlling the ordering for both the instantiations of the grouping variables and the aggregation result. For example, if the sorting specification is `asc` then first we will get the results that are grouped around the smallest instance of `Gs`, and those results themselves will be ordered in the ascending order. On backtracking, we get the results grouped around the next-smallest instantiation of `Gs` and, again, those results will be ordered in the ascending order.

The third form of the above `setof/bagof` aggregates provides the most general means of controlling the ordering for the case when grouping is used. Here *SortSpec1* is used to control the results and *SortSpec2* controls the grouping order. That is, the instances of `Gs` are first ordered according to *SortSpec2*, and then the results associated with the first grouping in the `SortSpec2`-ordering are returned. These results are ordered according to *SortSpec1*. Then the next grouping in the *SortSpec2*-ordering is chosen and its associated results are returned according to *SortSpec1*-ordering, etc. Both *SortSpec1* and *SortSpec2* can be the simple sort specs `asc` or `desc` or they can be lists of the fine-grained specs of the form `asc(`*i*`)` or `desc(`*i*`)`.

Example:

```
q(a,a,r(z,1)). q(a,a,r(g,2)). q(b,b,r(b,2)). q(b,b,r(d,3)).
q(b,b,r(d,3)). q(b,c,r(d,3)). q(b,c,r(e,1)). q(e,e,r(k,3)).
q(e,e,s(k,4)). q(e,e,s(p,5)). q(e,e,s(p,5)). q(e,e,s(p,5)).
```

```
?- ?Y = setof{?Z[?X,?Q]([asc(1),desc(2),desc(3)],[desc(1),asc(2)])|q(?X,?Q,?Z)}.
```

will produce this result:

```
?Y = [r(d,3), r(b,2)]
?X = b
?Q = b

?Y = [r(e,1), r(d,3)]
?X = b
?Q = c

?Y = [r(k,3), s(p,5), s(k,4)]
?X = e
?Q = e

?Y = [r(z,1), r(g,2)]
?X = a
?Q = a
```

Here the results are sorted as follows:

- ascending order of the functors of terms that bind `?Z` (i.e., `r` then `s`);
- descending order of the arguments of these terms (i.e., the arguments of `r(...,...)` and `s(...,...)`);
- descending order of the terms that bind `?X`;
- ascending order of the terms that bind `?Q`.

30.5 Aggregation and Set-Valued Methods

Aggregation is often used in conjunction with set-valued methods, and *FLORA-2* provides several shortcuts to facilitate this use. In particular, the operator `->->` collects all the values of the given method for a given object in a set. The semantics of these operators can be expressed by the following rules:

```
?O[?M->->?L] :- ?L=setof{?V|?O[?M->?V]}

?O[|?M->->?L|] :- ?L=setof{?V|?O[|?M->?V|]}
```

Note that `?L` in `?O[?M->->?L]` and `?O[|?M->->?L|]` is a list of oids.

The special meaning for `->->` implies that this construct *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, FLORA-*2* provides another the primitive `+>>`. A statement of the form `o[m+>>s]` is true if the list of all values of the attribute `m` for object `o` *contains* every element in the list `s`. A statement of the form `o[|m+>>s|]` is true if `s` is the list of all values of `m` on the object `o`, which are obtained by inheritance from the superclasses of `o`.

For instance, the following query tests whether all Mary's children are also John's children:

```
?- Mary[children->->?L], John[children+>>?L].
```

As with `->->`, the use of `+>>` is restricted to rule bodies.

31 Control Flow Statements

FLORA-*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.

31.1 If-Then-Else

This is the usual conditional control flow construct supported by most programming languages. For instance,

```
?- \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 FLORA-*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 knowledge engineer wants such a conditional to 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 $\mathcal{F}$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 \false`. 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.

The form `if-then` also has the following alternative forms: `~~>`and `<~~`.

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 46.2).

31.2 Yet another If-Then-Else

There is a faster version of If-Then and If-Then-Else, but it is permitted only if the If-condition is a builtin primitive, a non-tabled Prolog predicate, or a predicate/method prefixed with the %-sign. In that case, the if-statement can be written as

if-condition `-->>` *then-part*

Apart from the restrictions on the if-condition, this differs from `\if` *if-condition* `\then` *then-part* in that `-->>` is false if *if-condition* is false (regardless of *then-part*) while `\if` *if-condition* `\then` *then-part* is true/false if and only if *then-part* is true/false.

The fast If-Then-Else version has this form:

(*if-condition* `-->>` *then-part* ; *else-part*)

If has the same semantics as `\if` *if-condition* `\then` *then-part* `\else` *else-part*, but is faster and has restrictions on *if-condition*, as explained above.

31.3 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.

31.3.1 The while-do and do-until Loops

These loops are similar in intent to those in C, Java, and other procedural languages. 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 them 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), it is not guaranteed 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. Examples:

```
p({1,2,3}).
?- \while p(?X) \do writeln(?X)@\plg.
?- \do (p(?X),writeln('loop executed for'=?X)@\io) \until \naf p(?X).
```

For the second loop above, note that `?X` must be bound at the time the until-condition is checked. When negating conditions under `\until`, caution should be exercised. For instance, something like

```
?- \do writeln('loop executed')@\io \until \naf p(?X).
```

would not have worked: the loop would execute only once because `\naf p(?X)` will succeed on the first check (since `?X` is unbound, `\naf p(?X)` is undefined, and in loop conditions "undefined" is treated as a successful test).

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.

Subtleties related to the loop conditions. In both of the above loops, `Condition` should normally be user-defined backtrackable predicate. The use of non-backtrackable predicates as conditions requires special care, as described below.

First, not all non-backtrackable Prolog predicates fail when their work is done. For instance, `file_read_line_atom/2`, which reads files line-by-line, fails at the end, but `read/1` and `read/2` do not. Instead, they return the atom `end_of_file` when done. Therefore, `read(?X)@\prolog` or `read(?X)@\io` can never be a condition in the above loops; use `(read(?X)@\prolog, ?X \== end_of_file)` instead.

Second, non-backtrackable predicates, like `file_read_line_atom` and `read`, will fail if they are backtracked over. Therefore, if they are used as conditions in the `while-do` or `do-until` loops, they will execute only once. Fortunately, $\mathcal{F}$LORA-*2* provides a way to use such predicates in the above loops by wrapping them with the `\repeat` hint. For instance (where we use $\mathcal{F}$LORA-*2*'s equivalents for `file_read_line_atom`, `open`, and `close`),

```
?- myfile[open(read) -> ?Stream]@\io,
    \while \repeat(?Stream[readline(atom,?Line)]@\io) \do writeln(?Line)@\io,
    ?Stream[close]@\io.
```

Third, for the `do-until` loop, the loop quits immediately after the condition becomes true. Therefore, if one wants to use non-backtrackable predicates like `file_read_line_atom` or `read` (or their $\mathcal{F}$LORA-*2* equivalents like `?Stream[readline(atom,?Line)]@\io`) then they must be negated. For instance,

```
q({1,2,3,4,5}).
?- myfile[open(read) -> ?Stream]@\io,
    \do (q(?X),writeln('loop executed for X'=?X)@\io)
    \until \repeat(\+ (?Stream[readline(atom,?Line)], writeln(?Line))@\io),
    ?Stream[close]@\io.
```

If `myfile` has 5 or more lines, this loop will execute 5 times and then fail. If `myfile` has less than 5 lines, the loop will execute once per line and then succeed.

Observe that `\naf` should *never* be used with non-logical conditions like the above, especially if these conditions are called with unbound variables.

31.3.2 The while-loop and loop-until Loops

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 in `while-loop` loops and until it becomes true in `loop-until` loops. 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`) or true, in case of `loop-until`.

As in the case of the previous two loops, `while-loop` and `loop-until` succeed even if `Condition` is false (`while-loop`) or is true (`loop-until`) right from the outset. The only case when these loops fail is when `Action` fails — see below for the 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.

Subtleties related to the while-loop and loop-until 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 continue forever.

Also, keep in mind that in any of the four loop statements, if `Action` fails, the loops terminate and are evaluated to *false*. 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 are no more ways to backtrack. However, `while-loop` and `loop-until` have 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 runs forever. Therefore, as mentioned above, the `while-loop/loop-until` loops must make sure that `Condition` is modified by `Action`. Thus, if `Action` has non-transactional updates, the user must arrange 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 not terminate, 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 this idiom is useless if the action is expected to always succeed.

32 Constraint Solving

*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). To pass a constraint to a constraint solver in the body of a *F*LORA-*2* rule (or query), simply include it inside curly braces.

Here is a 2-minute introduction to CLPR. Try the following:

```
?- insert{p(1),p(2),p(3)}.
?- ?X>1, ?X<5, p(?X).
```

Traditional logic languages, like Prolog, would give an error in response to this query. *F*LORA-*2* is actually pretty good in this respect, as it will delay the inequalities until they can be solved. So, it will return two answers: 2 and 3. But what if we ask *F*LORA-*2* to solve an equation:

```
?- insert{p(1),p(2),p(3),p(9)}.
?- ?X=?Y*?Z, ?Y>1, ?Z>2, p(?X), p(?Y).
```

That *F*LORA-*2* cannot do without the help of *Constraint Logic Programming*. Constraint logic programming takes the view that `?X=?Y*?Z, ?Y>1, ?Z>2` is a a *constraint* on the set of solutions of the query `p(?X)`. This approach allows Prolog to return meaningful answers to the above query by solving this constraint. However, the user must explicitly tell *F*LORA-*2* which view to take: the traditional view that treats arithmetic built-ins as infinite predicates or the one that treats them as constraints. This latter view is indicated by enclosing constraints in curly braces. Thus, the above program becomes:

```
?- insert{p(1),p(2),p(3),p(9)}.
?- {?X=?Y*?Z, ?Y>1, ?Z>2}, p(?X), p(?Y).

?X = 9
?Y = 2
?Z = 4.5000

?X = 9
?Y = 3
?Z = 3.0000
```

Note the use of curly braces in the above example: they are essential in order to tell the system that you want constraints to be *solved* rather than *checked*.

33 Low-level Predicates

Unadulterated Prolog predicates. Sometimes it is useful to define predicates that are handled *directly* by the underlying Prolog engine. Such predicates would be represented as Prolog, not HiLog predicates. They are visible to HiLog queries, are not tabled automatically, and they are indexed as any other Prolog predicate. One use of such predicates, as *sensors*, is described in Section 34. To support this feature, $\mathcal{F}$LORA-*2* provides two directives:

```
:- prolog{predname/arity, predname/arity, ...}.
:- table{predname/arity, predname/arity, ...}.
```

The first directive defines a predicate of a certain arity as a Prolog predicate to be handled directly by Prolog. If this predicate is defined recursively, it might need to be tabled to help with termination or to reduce computational complexity. This is accomplished with the help of the second directive. The `table` directive implies the `prolog` directive, however, so there is no need for the former in that case.

As mentioned above, predicates declared with the `prolog` directive are not visible to HiLog queries. For instance, in

```
:- prolog{foo/2}.
?- ?X(?,?).
```

the variable `?X` will *not* be bound to `foo`.

The names of the predicates defined via the `prolog` directive are scrambled so there is no fear of a clash with Prolog builtins or `:- prolog{...}`-defined predicates in other $\mathcal{F}$LORA-*2* modules.

At present prolog predicates are automatically exported and there is no way to encapsulate them in a module. To refer to a prolog predicate defined in a different module, e.g., predicate `foo/1` in module `bar`, the following idiom can be used:

```
:- prolog{foo/1}.
... ... ...
... :- ..., foo(?X)@bar, ...
```

Note that there are no *executable* `prolog` or `table` directives (i.e., invocable via ?– or from the $\mathcal{F}$LORA-*2* shell).

The `prolog` and `table` directives in multi-file modules. Recall from Section 17.5 that a module can consist of several files: the first file would be normally loaded into the module and the subsequent files are added. Once a `prolog` or `table` directive is issued, it affects the compilation of the corresponding predicate symbols. The files added after the file that contains the initial `prolog/table` directives *must* include the necessary `prolog` directives, *if* that uses the affected predicate symbols. If this is not done, the system would issue an error, to prevent hard-to-find error from creeping into the knowledge base.[18] It should be noted, however, that if a file with `prolog` declarations (e.g., `:- prolog{p/1}` or `:- table{p/1}`) is loaded into a module, `foo`, that declaration is automatically made available to the FLORA-*2* shell. So, such predicates are readily accessible from the shell (e.g., as `p(?X)@foo`).

The `\nontabled_module` directive. For simple modules that *do not have recursive rules*, significant amount of main memory can be saved by telling FLORA-*2* to not table predicates and methods. This is especially useful for modules that process large amounts of data and can be done by placing the following directive in the module file:

```
:- \nontabled_module.
```

34 Sensors: Predicates with Restricted Invocation Patterns

Sometimes it is useful to define predicates that have fixed invocation patterns: the requirements that certain arguments must be bound (non-variable) or be ground. FLORA-*2* provides special support for this kind of predicates, which are called ***sensors***. Namely, if a predicate is registered as a sensor, FLORA-*2* will monitor the binding pattern of that predicate and, if the predicate is called before the binding conditions are fulfilled, it will delay the predicate until the conditions are fulfilled. If at some point FLORA-*2* determines that the binding condition *cannot* be satisfied, FLORA-*2* will call the sensor anyway. The sensor's implementation can then examine the state of the argument bindings and issue an error, if appropriate.

The overall scenario for sensor use is as follows:

- Sensors are *used* and *defined* in separate files. Sensors must also be *declared*. Declaration is done via the `defsensor` directive; definitions are done by means of the regular rules.
- A sensors can be defined in a `.P` file using the Prolog syntax (P-sensors) or in `.flr` files using the FLORA-*2* syntax (F-sensors). P-sensors can also be defined as external modules in the C language. Usually all sensor declarations are collected in one `.flr` file, which should be

[18] Note: subsequently added files need only the `prolog` directives—even for predicates that are declared with a `table` directive.

loaded into a separate FLORA-2 module. That `.flr` file should also contain the rules that define F-sensors.

- FLORA-2 files containing sensor definitions *cannot* be added to another FLORA-2 module (e.g., loaded using the `add{...}` command).
- There are no restrictions on how sensors are to be defined. For instance, they can be recursive. However, there are certain conventions to abide by for P-sensors.
- Sensors are *used* in modules *other* than those where they are defined. To *use* a sensor in a file, it has to be declared in that file with the `usesensor` directive.

  However, if a file with a `usesensor` declaration is loaded into some module, all files that are *compiled* in the *same* FLORA-2 session and then *added* into the same module will inherit that declaration. In that case, no explicit `usesensor` declaration is needed. However, if the file being added is compiled in a different FLORA-2 session (which does not have the requisite `usesensor` declarations loaded) then explicit `usesensor` declarations are required.
- FLORA-2 also provides executable versions of the directives `defsensor` and `usesensor`.

In most cases, the developer would choose F-sensors. There are two slight advantages in choosing to define a sensor as a P-sensor, however: the encapsulation provided by Prolog modules or if the sensor is implemented completely in C.

Declaring a sensor. There are two forms of the `defsensor` directive. The first is used for P-sensors (defined in `.P` files in Prolog or in `.c` files in C); the second is used for F-sensors (defined using FLORA-2 syntax in `.flr` files).

```
:- defsensor{sens1(?Y,?X), sensorfoo, (nonvar(?X),ground(?Y))}. // P-sensor
:- defsensor{sens2(?Y,?X), (ground(?X),ground(?Y))}.            // F-sensor
```

The first argument is a sensor invocation template. The last argument is a *guard*: a Boolean combination of `nonvar/1` and `ground/1` predicates applied to the input variables of the sensor. The sensor will be *delayed* by the FLORA-2 engine until the guard is satisfied (or until the engine determines that the guard cannot be satisfied).

In the first `defsensor` directive, which applies only to P-sensors, the middle argument is the name of the Prolog module in which the P-sensor is defined. In our case, `sens1` is declared as a P-sensor in the Prolog module `sensorfoo`. In that case, FLORA-2 will expect those rules to be in a file named `sensorfoo.P` (or `sensorfoo.c`, if the sensor is defined as an external C module) and that file should be found somewhere on the FLORA-2 search path (e.g., in the current directory).

Defining an F-sensor. The F-sensor `sens2` above is declared using a two-argument `defsensor` directive, so its definition is expected to be in a `.flr` file and then loaded into a separate *FLORA-2* module. Here is an example of such a definition:

```
sens2(abc,cde) :- !, writeln(details=\?F+\?L)@\io.
sens2(?X,?Y) :-
     \if \+ground(?X) // only ?X must be ground
     \then
        abort(['In file ', \?F, ', line ', \?L, ': ',
                'Instantiation error in arg 3 in sens2'])@\sys,
     ?Y \is ?X/2,
     writeln(Y=?Y)@\prolog,
     sens2(abc,cde).
```

The above rules are quite standard except for the quasi-variables `\?F` and `\?L`. When these quasi-variables occur in the body of a sensor definition, `\?F` is replaced with the file name from which the sensor was called; the quasi-variable `\?L` is replaced with the line number in the file where the call to the sensor occurred. A typical use of these quasi-variables is to report errors or issue warnings in case the sensor guard is not satisfied at the time the sensor is called, as illustrated in the above example. If these quasi-variables are used outside of a sensor definition, they are treated as new unbound variables.

Defining an P-sensor. From the perspective of the *user*, the above P-sensor `sens1` is a binary predicate. However, from the perspective of the *developer*, a P-sensor has two "hidden" extra arguments. These arguments are automatically pre-pended by the compiler to the list of the arguments that the user specified in a `defsensor` directive.

In any actual call (in a .flr file) to a P-sensor, the first argument will *always* be bound (at compile time) to the file in which the sensor is used and the second argument will be bound to the line number on which the call to the sensor occurs. This is done to make it possible to define sensors so that they would issue useful runtime errors to enable the user to quickly locate the offending call. Thus, in the actual rules that define a P-sensor, the first two arguments to the sensor must be distinct variables, and this is the responsibility of the P-sensor developer. The remaining arguments are up to the developer to choose.

The upshot of the above is that to define `sens1` one uses a predicate of arity 4, not 2. Here is an example of a definition for `sens1` in a Prolog file `sensorfoo.P`. As we have just explained, the actual predicate to be defined here must be `sens1` and its first two arguments must be distinct variables reserved for file names and line numbers (Prolog variables `F` and `L`).

```
:- export sens1/4.
```

```
sens1(F,L,X,Y) :-
     (var(X)       // X must be nonvar
     ->  abort(['In file ', F, ', line ', L, ': ',
                'Instantiation error in arg 1 in in sens1/2'])
     ; \+ground(Y) // Y must be ground
     ->  abort(['In file ', F, ', line ', L, ': ',
                 'Instantiation error in arg 2 in sens1/2'])
     ),
     Z \is X+Y,
     writeln(z=Z).
```

It should be noted that F-sensors are also internally represented by predicates that have two extra arguments. However, the transformations that add these arguments are done by the $\mathcal{F}$LORA-*2* compiler completely transparently to both the users and the developers of F-sensors.

Using a sensor. When sensors are used in $\mathcal{F}$LORA-*2* modules (which are different from the modules where the sensors are defined), they must be declared using the `usesensor` directive. For instance,

```
:- usesensor{sens1/2, sens2/2}.
?- sens1(?X,?Y), ?X=123, ?Y=aaaaa, sens2(?X,?Y), ?Y=345, ?X=bbbbb.
```

Note that if the definition of a sensor is recursive, tabling might be needed in order to ensure termination. In a `.P` file, this is done using the usual Prolog `table` directive. $\mathcal{F}$LORA-*2* provides a similar directive for sensors defined in `.flr` files:

```
:- table{sens2/2}.
```

This directive is to be placed in the file that contains the rules *defining* the sensor and the number of arguments must match the number used in the F-sensor declaration. It must come *after* the `defsensor` directive. For P-sensors, the arity must match the number of arguments used in the sensor definition, i.e.,

```
:- table{sens1/4}.
```

Sensors are viewed as low-level non-logical predicates, analogous to builtins, so they are not visible to HiLog queries. For instance, neither

```
?- ?X(?,?,?,?).
```

nor

```
?- ?X(?,?).
```

would bind `?X` to `sens1` or `sens2` (cf. Section 33).

35 Rearranging the Order of Subgoals at Run Time

Sometimes certain subgoals should be executed only if certain arguments are bound or ground and should be delayed otherwise. The reason for this might be correctness or efficiency of the execution. Since it may be impossible to know at compile time when the requisite arguments become bound, one might want to place the relevant subgoals as early as possible to the left and then wait until the binding conditions become true. Once they become true, the affected subgoals can be executed. It the binding condition never becomes true, then we can have two possible actions: abort or execute the subgoals anyway.

FLORA-2 supports this mode of goal rearranging via *delay quantifiers*. A delay quantifier has one of these forms:

`must`(*Condition*)
`wish`(*Condition*)

where *Condition* consists of the terms of the form `ground`(?*Var*) and `ground`(?*Var*) connected with "," (or `and`) and ";" (or `or`). If parentheses are not used in *Condition*, then the comma is considered to bind stronger than the semicolon.

Delay quantifiers are connected via the operator `^` to the actual goal that is to be executed with possible delay. The goal can be a simple frame or a predicate, or it can be a more complex goal of any kind that is allowed to appear in the rule bodies. In the latter case, the goal has to be enclosed inside parentheses. A subgoal with an attached delay quantifier is called a *delayable subgoal* and the regular subgoal part in a delayable subgoal is said to be *controlled* by the quantifiers. So, the general syntax is

delay-quantifier`^`*delay-quantifier*`^`...`^`*Goal*

Delayable subgoals can be nested and multiple delay quantifiers can be attached to the same goal. For instance,

```
must(ground(?X) or nonvar(?Y))^?X[foo->?Y]
wish(ground(?X) and ground(?Y) ; nonvar(?Z))^(p(?X), foo[bar(?Y)->?Z])
wish(ground(?X) or ground(?Y))^must(nonvar(?Z))^
           (p(?X), wish(nonvar(?W))^q(?W), foo[bar(?Y)->?Z])
```

When a delayable subgoal is to be executed, the attached delay quantifiers are checked. If at least one the quantifiers is not satisfied (i.e., if its condition is not satisfied) then the goal is *not*

executed, but is delayed instead until such time that all the quantifiers are satisfied or the engine determines that satisfying all the quantifiers is impossible. In the above example, the first goal would be delayed is `?X` is non-ground and `?Y` is an unbound variable. The second delayable subgoal has a more complex execution condition and it controls a more complex subgoal, the conjunction of `p(?X)` and `foo[bar(?Y)->?Z]`. This subgoal will be delayed if either `?X` or `?Y` is non-ground and `?Z` is an unbound variable. The third delayable subgoal is even more complex. First, it involves two delay quantifiers; second, it controls a complex subgoal that is a conjunction of three other subgoals, and one of them is itself a delayable subgoal. This complex delayable subgoal will be executed immediately only if `?X` or `?Y` are ground and `?Z` is not a variable. Otherwise, it will be delayed. But when the controlled subgoal is eventually ready for execution, its middle part, `wish(nonvar(?W))^q(?W)`, might still be delayed if `?W` is an unbound variable.

When a delayable subgoal is delayed, its attached quantifiers are periodically checked for satisfaction. If the first (leftmost) quantifier is satisfied, the controlled subgoal is executed. If this subgoal still has a controlling delay quantifier, then this quantifier's condition will be checked, etc. At some point during the execution, the inference engine might stumble upon a delayed subgoal and *determine* (provably!) that its first delay quantifier is *not satisfiable* any more, i.e., subsequent execution will not be able to bind the variables in that quantifier in the required way. In this case, the cause of action depends on the type of the quantifier. If the unsatisfied quantifier is a `wish`-quantifier, then the controlled subgoal is executed anyway: the unsatisfied delay condition was only a "wish." If, however, the unsatisfied quantifier is a `must`-quantifier, the execution of the subgoal is aborted and the error message will indicate which `must`-condition was at fault. To illustrate this on an example, consider the following delayable subgoal issued as a query against a knowledge base that contains the listed facts:

```
p(1).
foo[bar(2)->3].
?- must(ground(?X) or ground(?Y))^wish(nonvar(?Z))^
            (p(?X), must(nonvar(?W))^q(?W), foo[bar(?Y)->?Z]),
   ?X=1.
```

When the subgoal is first encountered, it is delayed because `?X` and `?Y` are not ground. Then the engine executes the subgoal `?X=1` and `?X` gets bound to 1. The delay condition becomes satisfied, so the engine considers the subgoal

```
wish(nonvar(?Z))^(p(?X), must(nonvar(?W))^q(?W), foo[bar(?Y)->?Z])
```

Clearly, `?Z` will not be bound if the engine continues delaying the controlled subgoal, so it must decide on the final disposition for this delayable subgoal. Since the delayable subgoal is controlled by a `wish`-quantifier, the subgoal is executed despite the fact that the delay condition does not

hold. During the execution, `p(1)` is satisfied, `must(nonvar(?W))^q(?W)` gets delayed, and then `foo[bar(?Y)->?Z]` is satisfied. Now the disposition of the remaining delayed subgoal has to be decided. Since `?W` cannot be bound any more and the controlling quantifier is of the `must`-variety, the engine will throw an error:

```
++Abort[Flora-2]> in file foo.flr, line 3: unsatisfied must-condition.
Goal: q(?A)
Condition: nonvar(?A)
```

Forcing immediate execution of delayed subgoals. Sometimes it is necessary to tell the system to stop delaying subgoals and force their immediate execution. For instance, in

```
?- must(ground(?X) or ground(?Y))^wish(nonvar(?Z))^
                  (p(?X), wish(nonvar(?W))^q(?W), foo[bar(?Y)->?Z]),
    ...,
    ...,
    test(?X,?Y,?Z,?W).
```

the query `test` might be executed before the queries `p(?X)`, `q(?W)`, and `foo[bar(?Y)->?Z])`, *if* some of the delay quantifiers remain unsatisfied. If instead we want to make sure that those queries are executed before `test(?X,?Y,?Z,?W)`, FLORA-*2* provides the *immediate execution operator*, `!!`, which forces immediate execution of all the "reachable" delayed subgoals:

```
?- must(ground(?X) or ground(?Y))^wish(nonvar(?Z))^
                  (p(?X), wish(nonvar(?W))^q(?W), foo[bar(?Y)->?Z]),
    ...,
    ...,
    !!,
    test(?X,?Y,?Z,?W).
```

In this example, the aforesaid queries will be executed before `test(?X,?Y,?Z,?W)`.

A delayed subgoal, G, is *reachable* by an occurrence of the immediate execution operator `!!` if G is suspended on a ground/nonvar condition that involves a variable that appears in the the scope of the rule or the query where the aforesaid occurrence of `!!` is found. What it means to be suspended on a variable is a little trickier. In the above example, if neither `?X` nor `?Y` is ground, then `p(?X)`, `q(?W)`, and `foo[bar(?Y)->?Z])` would be suspended on either `?X` or `?Y`. If one of these variables becomes ground, then these queries would be suspended on `?Z`, if `?Z` is still not bound. Once `?Z` gets bound, then `p(?X)` and `foo[bar(?Y)->?Z])` will no longer be suspended, but `q(?W)` will still be suspended on `?W` unless `?W` is already bound by that time.

Note, however, that `!!` will not have any effect on subgoals that are suspended on variables that are not in the scope of the rule or query containing this immediate execution operator. For instance, in

```
p(?X) :- must(nonvar(?X))^foo(?X,?Y), wish(ground(?Z))^bar(?Z), moo(?Y,?X,?Z).
moo(?Y,?X,?Z) :- ..., !!, ...
?- p(?W), !!, writeln(?W)@\plg.
```

the variable `?W` is in scope for the query. Since `p(?W)` unifies with the head of the first rule, `?W` and `?X` become the same variable. This means that `foo(?X,?Y)` is reachable from `!!` and will be forced to execute by that operator. In contrast, `bar(?Z)` is suspended on the variable `?Z`, which is not in the scope of the query. Therefore, `bar(?Z)` will not be forced to execute by the operator !! in the above query. On the other hand, `?Z` is in the scope of the second rule above, so the occurrence of `!!` in *that* rule will force the execution of `bar(?Z)`.

36 Rule Ids and Meta-information about Rules

Every rule in *FLORA-2*—whether it appears in a file, is inserted at run time, or is reified and lives as an object it its own right—is assigned a unique *rule Id* and the user can also supply additional meta-data for it. The rule Id can be either given explicitly by the knowledge engineer or, if not given explicitly, is assigned by the system (at compile time for rules that live in files and at run time for inserted and reified rules).

A rule Id is a triple of the form $(local\_id, file\_name, module)$, where

- *local_id* is a term that is either explicitly given to the rule by the author or is generated by the compiler or loader.
- *file_name* is the *local* file name where the rule occurs. The name does *not* include the directory part, but it *does* include the name extensions (e.g., `foo.flr`). The `#include` commands are taken into account, so if a rule is found in file `foo` that is `#include`d in file `bar` then the rule Id will use `foo`, not `bar`.
- *module* is the module into which the rule is loaded.

When the file name and the module part of a rule Id is clear or immaterial, we will sometimes refer to the local part of a rule Id as simply an "Id."

For rules that are inserted dynamically at run time via an `insert{...}` or `insertrule{...}` statement, their file name is to be specified as `dynrule(`*containing-file*`)@\prolog`, where *containing-file* is the local file name of the file that contains the insert statement in question. This also applies

to rules found in files that are *added* (as opposed to loaded). For instance, consider the following file

```
// File: foo.flr
@!{id1} p1 :- q1.
?- insert{@!{id2} p2 :- q2}.
```

If this file is *loaded* into the module `bar` then the first rule will have the full Id `(id1,'foo.flr',bar)` and the second `(id2,dynrule('foo.flr')@\prolog,bar)`, but if `foo.flr` is *added* to `bar` then the first Id will be `(id1,dynrule('foo.flr')@\prolog,bar)` (and the second as before).

If a rule is inserted via the FLORA-*2* command line interface then the file component of the Id will be `dynrule(?)@\prolog`, i.e., the file name component is a variable. For instance, if a rule like the following is inserted at a command prompt

```
flora2 ?-   insert{(@!{id3} p3 :- q3)@bar}.
```

then its Id is `(id3,dynrule(?)@\prolog,bar)`.

FLORA-*2* supports the use of rule *descriptors* of the form

- *Rule Id descriptor*: `@!{id[frame]}` or just `@!{id}`,
  where `id` is a term and the frame information is the same as what is allowed in a simple (not nested) frame. Some properties in the frame have special meaning as detailed below.

  The rule Id descriptor is used to explicitly specify *only* the *local id* part of the rule Id, as described above. The other two components are assigned by the system. If no explicit rule Id is given, the system will generate one.

- `Tag descriptor`: The property `tag` indicates that the values of that property are rule tags, which are used in defeasible reasoning (Section 38). For instance,

  ```
  @!{abc[tag->{tag1,tag2},author->'kifer@cs.stonybrook.edu']} head :- body.
  ```

  Here the rule is given an explicit rule Id, `abc`, and two tags, the terms `tag1` and `tag2`. The property `author` (as most others) is not special; it can be used for whatever purpose the user decides to use it.
  Because rule tags are very common pieces of metadata, FLORA-*2* provides a convenient shortcut, which is used as a standalone descriptor. For instance, the following is equivalent to the previous descriptor:

  ```
  @{tag1,tag2} @!{abc[author->'kifer@cs.stonybrook.edu']} head :- body.
  ```

- *Defeasibility descriptor*: Defeasibility descriptors are used for defeasible reasoning, as described in Section 38. The descriptor `strict` means that the rule cannot be defeated by a higher-priority rule, and `defeasible` means it can be. These are Boolean properties in the descriptor frame and, as one may guess, only one of them can be specified for any given rule. For instance,

  ```
  @{tag1,tag2}
  @!{abc[author->'kifer@cs.stonybrook.edu', defeasible]} head :- body.
  ```

  Defeasibility descriptors are also fairly common and $\mathcal{F}$LORA-*2* provides shortcuts for them as well. As with tags, these shortcuts are specified as standalone descriptors. For instance,

  ```
  @{tag1,tag2}
  @@{defeasible}
  @!{abc[author->'kifer@cs.stonybrook.edu']}
  head :- body.
  ```

  Of course, we did not save any typing with the above defeasibility shortcut. However, this shortcut does improve the visibility of the fact that a particular rule is defeasible (or strict). Also, when no explicit Id and frame properties are specified and the user opts for the default rule Id, the defeasibility and the tag shortcuts save significant amount of typing. For instance,

  ```
  @{tag1,tag2} @@{defeasible} head :- body.
  ```

  Without these shortcuts, the user would have to use the *current rule Id quasi-constant*, `\@!`, which gets substituted for the Id of the rule in which it occurs:

  ```
  @!{\@![tag->{tag1,tag2}, defeasible]} head :- body.
  ```

  In this case, `\@!` gets replaced by the default Id, which the system assigns to this rule.

The descriptors extend the rule syntax as follows:

```
rule-descriptors Head :- Body.
rule-descriptors Fact.
```

Any number of tag descriptors is allowed but only one rule Id and one defeasibility descriptor. In other places, rule descriptors are either ignored or will cause syntax errors.

Rule tags and Ids are terms and so can have variables. For instance,

```
@{allow(?Device,?Person)}  authorized(?Person,?Device) :- \naf abused(?Person,?Device).
```

If no Id is given, a system-generated rule Id is used. If no tag is given, the Id of the rule also serves as a tag.

If no defeasibility descriptor is given then a rule is considered defeasible by default if an explicit tag is specified. Otherwise, the rule is considered strict. However, for defeasibility to have any effect, the rule must be in a *defeasibility-capable* document (local file or a remote document), i.e., the document must have the directive `:- use_argumentation_theory` at the top (see Section 38). Also, rules whose heads are transactional predicates or methods are currently never defeasible—no matter what the defeasibility descriptor says.

36.1 The Current Rule Id Quasi-constant

We have already seen a use of this quasi-constant earlier. In general, rules can reference their Ids even if the Id is not given explicitly by the author and thus cannot be determined by just looking at the rule. This is accomplished with the help of the special "current rule Id" quasi-constant `\@!`. This quasi-constant gets replaced by the *local part* of the Id of the rule where the quasi-constant occurs. The replacement happens at compile time for the static rules and at run time for dynamic or reified rules. To get the file name part, use `\@F` and the module part use `\@` — the quasi-constants introduced in Section 8.6.

36.2 Enabling and Disabling Rules

Rule Ids make it possible to *enable* and *disable* rules, producing the effect of deleting and re-inserting rules. The difference is, however, that enabling and disabling do not actually change the rules in the system (so they are very fast compared to `insert` and `delete`) and, most importantly, they apply to static rules, not just the dynamic rules. To specify a rule, one needs to specify the local Id of the rule in question, the file name of the rule, and its module. For dynamic rules, the file name is `dynrule(`*containing-file*`)`, where *containing-file* is the file that contains the `insert...` or `insertrule...` statement that inserted the dynamic rule. The following primitives are provided:

- `enable{`*LocalId,File,Module*`}`
- `disable{`*LocalId,File,Module*`}`
- `tenable{`*LocalId,File,Module*`}`
- `tdisable{`*LocalId,File,Module*`}`

- `isenabled{`*LocalId,File,Module*`}`
- `isdisabled{`*LocalId,File,Module*`}`

In the `enable`/`tenable` primitives, all arguments must be bound. The first two primitives are non-transactional, i.e., these operations will not be undone if the query in which they occur fails. The second two primitives, `tenable` and `tdisable`, *are* transactional. Their effects are undone on backtracking. For instance,

```
bbbb.
@!{r1} aaaa:-bbbb.
?- tdisable{r1,\@F,\@}, aaaa.
No
?- aaaa.
Yes
```

Here the first query disables the rule `r1` so `aaaa` becomes false and thus the entire query fails. Due to that, the disabling action is undone, so the rule remains enabled. Therefore, the second query succeeds.

The last two primitives are queries that tell us whether a particular rule is enabled or disabled. To enable or disable a rule, the rule must already exist as a static or dynamic rule. Enabling and disabling operations are idempotent: they always succeed and applying `enable` to an enabled rule or `disable` to a disabled rule has no effect. If a rule with the specified Id and module does not exist in the system, both `enable` and disable fail and so do both of the above queries.

All of the above primitives have a one-argument form, which is a shortcut for the three-argument version, where argument 2 is `\@F` and argument 3 is `\@`. For instance, `tenable{foo}` instead of `tenable{foo,\@F,\@}`.

The difference between disabling and deleting a rule. It is important to realize that disabling a rule, as described here, and deleting a rule, described in Section 28.3, are two very different operations, although there are similarities. For similarities, both operations take the affected rule(s) out of the reasoning process and the effect is as if the rule is "not there." The major differences are:

1. Rule deletion works only for dynamic rules (those added with `insertrule`), while disabling works for all rules.
2. A deleted rule is physically removed from the system. A disabled rule is only marked as disabled. It can be re-enabled by supplying its Id and module as arguments to the `enable` primitive. In contrast, to re-instate a deleted rule, it must be re-inserted, which requires

supplying the entire original rule as an argument. Disabling/enabling are very fast operations, while deleting/inserting a rule is much more expensive.

3. For *multi-headed* rules, there is another important difference. A multi-headed rule is one that has more than one literal in the head, which is equivalent to several single-headed rules. Such rules typically arise when a complex frame is put in a rule head. With a *deleterule* operation, one can delete just one of the heads, leaving the remainder of the rule intact. For instance, in

```
?- insertrule{?X:person[name->?N] :- pred(?X,?N)}.
?- deleterule{?X:person :- pred(?X,?N)}.
```

the `person[name->?N] :- pred(?X,?N)` part of the rule remains. In contrast, in

```
?- insertrule{@!{abc} ?X:person[name->?N] :- pred(?X,?N)}.
?- disable{abc,\@}.
```

the `disable`-primitive takes all parts of the rule out of circulation, while a subsequent matching `enable`-operation would reinstate the entire rule.

36.3 Changing the Defeasibility at Run Time

The defeasibility status of a rule can be changed via the following statements:

- makedefeasible{Id,File,Module}
- makestrict{Id,File,Module}
- isdefeasible{Id,File,Module}
- isstrict{Id,File,Module}

The last two primitives are queries. The primitive `makedefeasible` and `makestrict` are idempotent and always succeed. The queries `isdefeasible` and `isstrict` tell us whether the rule in question is currently defeasible or strict. If the rule with the given Id does not exist in the given module, both of these queries fail.

All of these primitives have a short, one-argument form, which is a shortcut for the three-argument version, where argument 2 is `\@F` and argument 3 is `\@`. For instance, `makedefeasible{foo}` instead of `makedefeasible{foo,\@F,\@}`.

36.4 Querying Rule Descriptors

Rule Ids and other rule descriptor data can be queried using the `clause{...}` and `@!{...}` constructs. The `clause{...}` construct was already introduced in Section 29. Now we introduce two new forms of this statement, which extend the statements defined in Section Section 29:

`clause{`*descriptors head,body*`}`
`clause{`*descriptors type,head,body*`}`

Here descriptors are rule descriptors as above and *type*, *head*, and *body* are as in Section 29. For instance,

```
?- clause{@{?Tag} @@{defeasible} foo[bar->?Val],?Body}.
```

This query will retrieve all defeasible rules whose head matches `foo[bar->?Val]`. The variable `?Tag` will be bound to the tag of the rule.

Note: The `clause{...}` primitive queries the rules as they were created by the author. Disabling a rule or changing its defeasibility status won't affect `clause...`. For instance, if we had a rule `@!{abc} @{defeasible} p:- q` and then this rule was disabled and/or made strict, `clause{@@{defeasible} p,q}` will still be true and `clause{@@{strict} p,q}` false. In this way, one can investigate the changes made to the status of the various rules in the course of time. □

The `@!{...}` construct is more convenient than the `clause`-construct and much more efficient, if one needs to query the *descriptor data only*, not the rule base. Inside the braces, this construct expects a usual frame formula of the form `ruleId[`$prop_1$`,`$prop_2$`,...]`, where $prop_i$ have the form `attr->val` or `term`. In the first case, the property is a regular attribute-value pair and in the second it is a Boolean property. Nested or composite frames are not allowed, but module specifications are supported, so one can query rule descriptors in another module. For instance,

```
?- @!{?R[foo->bar, tag->low, defeasible]}.
?- @!{rule123[]@foo}.
?- @!{?R[tag->high,strict,author->Bob]@foo}.
```

Recall that in all the above examples, only the local part of the rule Ids was queried (cf. `?R` in the queries above). To enable querying the file name and the module components of rule Ids, *FLORA-2* also provides the `file` and `module` properties. For instance,

```
?- @!{?R[foo->bar, tag->low, defeasible, file->?F, module->?M]}.
?- @!{rule123[file->?F, module->?M]@foo}.
?- @!{?R[tag->high,strict,author->Bob, file->?F, module->?M]@foo}.
```

In addition, the `type` property can be used to find out whether a particular annotated statement is a rule or a latent query (see Section 37):

```
?- @!{statement43[type->?T, module->main]}.
```

If the statement is a rule then `?T` will get bound to `rule`; if it is a latent query then this variable gets bound to `query`.

36.5 Reserved Properties in Rule Descriptors

The following property names in rule descriptors or in rule descriptor queries (described in Section 36.4) are reserved:

- `tag`
- `module`
- `file`
- `type`
- `defeasible`
- `strict`

The properties `module`, `file`, and `type` are not allowed in rule descriptors (i.e., in `@!{...}` in rule heads)—only in rule descriptor queries (i.e., in `@!{...}` and `clause{...}` in rule bodies)—because they are assigned at compile or load time.

37 Latent Queries

Latent queries are a cross between queries and rules. From queries they borrow the main purpose: to query the knowledge base. From rules they borrow the ability to use descriptors (so, latent queries have Ids and, possibly, other meta-data) and the fact that such queries are not executed immediately. Instead, they are saved to be called on-demand some other time. Like rules, latent queries can also be inserted, deleted, enabled, disabled, and queries.

Latent queries are useful as integrity constraints and also in GUI development, but because these queries can have arbitrary properties attached to them via descriptors, they can be used for various other, specialized needs (e.g., as standing queries that are called periodically to update

user views). At the same time, latent queries is an older mechanism, which is primarily retained for FLORA-2 and ERGO^{Lite} users. ERGO has newer, much more powerful mechanisms of *integrity constraints* and *alerts*.

A latent query has the following form:

descriptor `!-` *query-body*`.`

The descriptor part is mandatory and has the same form as the rule descriptors introduced in Section 36. The *query-body* part is the same as the body of a query. Thus, from regular queries the latent queries differ in that they have descriptors and they use `!-` instead of `?-`. From rules, this new type of queries differs in that they have no heads and use `!-` instead of `:-`.

The main difference with queries, however, is the fact that latent queries, when they appear in a file, are not executed right away. Instead, they are saved and must be called explicitly by their query Id using the `query{...}` primitive. In order to get results from such a query, the query Id must have variables. For instance,

```
p(1,2), p(2,3), p(3,4), p(2,4).
@!{test(?X,?Y)} !- p(?X,?Z), p(?Z,?Y).
?- query{test(?X,4),\@F,\@}.

?X = 1

?X = 2
```

(The quasi-constants `\@F`, `\@`, and others were introduced in Section 8.6.) Observe that latent queries are executed using their Ids. This is different from rule invocation, which is done using the rule-head as a query. In the above, `test(?X,4)` is *not* a predicate and it does not interfere with any other rule (even with the ones that have `test(?X,?Y)` as head- or body-predicates!).

The following two forms are equivalent:

```
?- query{test(?X,4),\@F,\@}.
?- query{test(?X,4)}.
```

However, the above will work only if `query{...}` is invoked from the same file as the one where the corresponding latent query is defined because `\@F` is specific to a file and `\@` is specific to a module. If the latent query is called from *another* file or from FLORA-2 shell (command line) then `\@F` cannot be used (and thus the 1-argument version of `query{...}` cannot be used either) because the quasi-constant `\@F` means "this file" and the files here are different (and the modules possibly

also). The file-component to use in the 3-argument version of `query{...}` in this case depends on where the latent query came from. This is described in detail for rules in Section 36 and the case of latent queries is similar:

- If a latent query of the form `@!{id123} !- query` was inserted (e.g., via `insertrule{...}`) into a module `bar` through the *FLORA-2* shell then the file component is `dynrule(?)@\prolog` and the invocation command for such a query would be

  ```
  query{id123, dynrule(?)@\prolog, bar}.
  ```

- If a latent query of the form `@!{id123} !- query` appears in a file with a local name `foo.flr` and this file is *added* to a module `bar` then the file component would be `dynrule('foo.flr')@\prolog` and the invocation command for such a query would be

  ```
  query{id123, dynrule('foo.flr')@\prolog, bar}.
  ```

- If a latent query of the form `@!{id123} !- query` appears in a file with a local name `foo.flr` and this file is *loaded* into module `bar` then the file component would be just `'foo.flr'` and the invocation command for the query would be

  ```
  query{id123, 'foo.flr', bar}.
  ```

Latent queries can be inserted, deleted, enabled, and disabled using the same primitives as the ones used for rules:

```
?- insertrule{@!{rr2(?X)[bar->4]} !- q(?X), ?X<4}.
?- deleterule{@!{rr2(?)[bar->?X]} !- ?}.
?- enable{qq1(?),'foobar.flr',}.
?- tenable{qq1(?),?F,?M}.
?- disable{qq2(?),F,}.
?- tdisable{qq2(?)}.
```

Finally, like rules, latent queries can be themselves queried using the primitive `clause...` and the meta-query @!.... In the clause... primitive, the head part is ignored.

```
?- clause{@!{?X[type->query]} ?,?B}.

?X = rr1(?_h4115)
?B = (${p(?_h4115)@main}, ?_h4115 < 3)
```

```
?X = rr2(?_h4156)
?B = (${q(?_h4156)@main}, ?_h4156 < 4)

flora2 ?-  @!{?X[type->query]}.

?X = rr1(?_h3033)

?X = rr2(?_h3046)
```

Note the use of the `type` property in the descriptors of the above two queries. This is a special builtin descriptor property, which can be used only in rule bodies. The value of that property is `query` for latent queries and `rule` for rules.

In $\mathcal{ERGO}$, latent queries are used for automatic maintenance of integrity constraints—a feature not available in $\mathcal{F}$LORA-*2*/$\mathcal{E}$RGO^{Lite}.

38 Defeasible Reasoning

Defeasible reasoning is a form of non-monotonic logical reasoning where some rule instances can be *defeated* by other rule instances. The defeated rules do not need to be satisfied by the intended model of the knowledge base. $\mathcal{F}$LORA-*2* supports the form of defeasible reasoning known as *Logic Programs with Defaults and Argumentation Theories*, as described in [16].

The basic mechanisms by which the user can control defeasible reasoning include the notions of rule *overriding* and literal *opposition*. We will not describe the semantics of defeasible reasoning formally, but instead will give an informal overview.

The basic idea is that the user specifies overriding and opposition using the appropriate predicates and then the *associated argumentation theory* decides which ground instances of which rules are defeated (and thus do not need to be satisfied by the intended model).

To this end, any rule can be *tagged* by adding a primitive @*tagname* to the left of the rule. Tags do not have to be unique and should not be confused with rule Ids. However, if no rule tag is explicitly given, the Id of the rule is used.

To see how it all works, consider a few examples. The first is a classical example that states that all birds generally fly, but some do not:

```
:- use_argumentation_theory.  // tell the system that rules can be defeasible
@{default} ?X:Flies :- ?X:Bird.        // all birds fly (by default)
@{penguin} \neg ?X:Flies :- ?X:Penguin. // but penguins don't
@{wounded} \neg ?X:Flies :- ?X:Wounded. // and neither do wounded things
```

```
Penguin::Bird.                              // penguins are birds
Sam:Penguin.                                // Sam is a penguin
Fred:Bird.                                  // Fred is a bird
Bob:{Bird,Wounded}.                         // Bob is a wounded bird
\overrides({wounded,penguin},default).      // being wounded or penguin
                                            // trumps the default rule
```

Here the first three rules have tags, which makes them defeasible. That is, whatever they might derive may still be invalidated. How? For instance, all birds fly by penguins do not. `Sam` is a penguin and thus a bird. According to the `default` rule, he should be able to fly, but according to the `penguin` rule he should not—a contradiction. The situation with `Bob` is similar: as a bird, he should be able to fly, but not as a wounded being. The way out is given by the last `\overrides` statement, which says that if a `default` rule's conclusion conflicts with a conclusion made via the `penguin` or a `wounded` rule, the latter take precedence. In other words, no contradiction occurs and both `\neg Sam:Flies` and `\neg Bob:Flies` are true (but not `Sam:Flies` or `Bob:Flies`). On the other hand, no conflict exists with respect to `Fred` and `Fred:Flies` is derived. What would have happened if the above conflicts were not resolved by the `\overrides` statement? In that case, both `Sam:Flies` *and* `\neg Sam:Flies` would be false, and similarly for `Bob`.

In the previous example, we do not have the `\opposes` statement, which was mentioned earlier. This is because the conflicting nature of any statement, like `Sam:Flies` and its `\neg`-negation is obvious. In some cases, however, such a conflict cannot be assumed a priori, and this where the `\opposes` statement helps. Consider a pricing database where the prices o items depend on the supplier:

```
:- use_argumentation_theory.
@{food} bread[price(store1) -> 2].
@{food} bread[price(store2) -> 3].
@{food} milk[price(store3) -> 2].
@{food} milk[price(store4) -> 2].
@{food} carrots[price(store2) -> 1].
@{food} carrots[price(store3) -> 1.5].
```

There is nothing strange in prices being different in different stores, but in some countries certain staple foods, such as bread, are regulated and must have the same price. There are many ways to deal with this problem. One would be to abort insert operations if they violate the same-price constraint. However, this is not always possible or desirable (e.g., when information is integrated from different sources). Defeasibility provides a different solution, which allows $\mathcal{F}$LORA-*2* to *tolerate* contradictions. What we can do is to say that it is a contradiction for a regulated item to have two prices:

```
\opposes(?Itm[price(?)->?P1],?Itm[price(?)->?P2]) :-
         regulated(?Itm),
         ?P1 != ?P2.
regulated({bread,milk}).
```

In the previous example, we provided a way to resolve a contradiction, but here we did not. As a result, the facts `bread[price(store1)->2]` and `bread[price(store2)->3]` will be false (i.e., the conflicting information will be zapped) while the price of milk will not be zapped, since there is no variation in prices in different stores. For carrots, the price varies, but it is not a regulated product, so no contradiction arises.

38.1 Concepts of Defeasible Reasoning

The previous section illustrated the concepts of overriding and opposition used in defeasible reasoning. We will look at these and other related concepts in more detail here.

A rule instance ρ_1, tagged i_1, *opposes* another rule instance, ρ_2 tagged i_2, if and only if `\opposes(`i_1`,`$head(\rho_1)$`,`i_2`,`$head(\rho_2)$`)` is true, where $head(\rho_i)$ denotes the head-literal of the corresponding rule. Literals for the form `lit` and `\neg lit` always oppose each other so this fact does not need to be explicitly specified using `\opposes`. A rule instance ρ_1, tagged i_1, *overrides* another rule instance ρ_2, tagged i_2, if and only if `\overrides(`i_1`,`$head(\rho_1)$`,`i_2`,`$head(\rho_2)$`)` is true. (For convenience, FLORA-*2* also defines 2-argument versions of these predicates, but this is unimportant for the present discussion.) We say that two rule instances are in *conflict* if they oppose each other and their bodies are true in the intended model of the knowledge base. A more detailed description of rule conflict and overriding appears in Section 38.3.

A rule can be defeated in several different ways. It can be

- *refuted*
- *rebutted*
- *disqualified*.

Different argumentation theories may assign different meanings to these concepts, but roughly they mean the following. A rule instance ρ_1, tagged i_1, *refutes* another rule instance ρ_2, tagged i_2, if and only if these rules are in conflict and the former rule instance overrides the latter. The rule instance ρ_1, tagged i_1, *rebuts* the rule ρ_2 with tag i_2 if and only if the two instances are in conflict and neither instance is refuted by some other rule instance.

Rule disqualification is not so commonly agreed upon a notion as the other two. A rule is *disqualified*, if it is *canceled*, i.e., if it matches a cancellation literal that is true and not defeated. More

details on cancellation appear in Section 38.4. More details on cancellation appear in Section 38.4. In addition, some argumentation theories disqualify rules in other cases as well. The default argumentation theory in *FLORA-2* postulates that a rule instance ρ_1, tagged i_1, is disqualified if

- If it transitively defeats itself, i.e., there is a sequence of tagged rule instances (i_1, ρ_1), (i_2, ρ_2), ..., (i_{n-1}, ρ_{n-1}), (i_n, ρ_n), such that $i_n = i_1$, $\rho_n = \rho_1$, and at each step (i_i, ρ_i) either refutes or rebuts (i_{i+1}, ρ_{i+1}).

These notions are very useful for debugging the knowledge base (finding out why certain inferences were or were not made), and we will discuss the corresponding debugging primitives in Section 38.8.

It is important to keep in mind that both `\overrides` and `\opposes` are user-level predicates, which are defined by the user as a set of facts (most commonly) and (less commonly) rules. The system uses these predicates to determine which rules are defeated but the user normally does not query these predicates explicitly except for debugging. These predicates are typically queried by the system to get answers to questions like "is there a rule that overrides (or opposes to) a given rule?" (e.g., `\overrides(?X,r123)`) or "check that this rule is not overwritten (or is opposed to) by some other rule" (e.g., `\naf \overrides(?X,r345)`). In other words, `\overrides` and `\opposes` are likely to be called with some of the arguments unbound. Because of that, one must be careful with what is in the body of the rules that define these predicates. For instance, in the following rule for `\overrides`, one parametrized rule-tag (as we shall see, rule tags can be terms with variables) is said to override another based on the values of the parameters, which are assumed to be numbers:

```
\overrides(rule1(?h),rule2(?j)) :- ?h >?j.
```

The problem here is that $>$ is a mode-sensitive predicate, which expects both of its arguments to be bound to an integer. However, as discussed above, `\overrides` might be called with one of the arguments, say the first, unbound. In this case, `?h` will be unbound at the time `?h >?j` is called and a runtime error will ensue.

38.2 Specifying an Argumentation Theory to Use

FLORA-2 supports a generalized form of defeasible reasoning. The user can enable defeasible reasoning on a per-module basis, and different theories of defeasibility can be used in different modules. The type of defeasible reasoning to be used depends on the chosen *argumentation theory*, which defines the *arguments* (as in "arguing") that the reasoner has to use in order to decide what inferences are to be defeated. Syntactically, defeasible reasoning is requested by placing one of the following directives at the top of the appropriate module (before any rules are given):

```
:- use_argumentation_theory.
:- use_argumentation_theory{Module}.
```

The first instruction directs $\mathcal{F}$LORA-*2* to use the default argumentation theory module, `\gcl`, which stands for Generalized Courteous Logic (or GCL). One can use a different theory of defeasible reasoning by implementing an appropriate argumentation theory and loading its file into some module `foo`. A file that actually uses this argumentation theory should have the directive

```
:- use_argumentation_theory{foo}.
```

at the top. Thus, different modules of the same $\mathcal{F}$LORA-*2* knowledge base can use different theories of defeasible reasoning. For instance, if some argumentation theory is implemented in file `myargth.flr`, then it can be declared in a $\mathcal{F}$LORA-*2* module using

```
:- use_argumentation_theory{foo}.
?- [myargth>>foo].
```

Defeasible theories must use certain API, which will be described in a later version of this manual. Meanwhile, one can construct such theories by analogy with GCL—see `AT/flrgclp.flr`.

Note that argumentation theories affect only defeasible rules and the defeasibility status of a rule can be changed from `strict` to `defeasible` and vice versa—see Section 36.

38.3 Rule Overriding and Conflicts

Rule overriding can be specified via one of the following two forms:

```
\overrides(RuleLab1,RuleLab2).
\overrides(RuleLab1,AtomForm1,RuleLab2,AtomForm2).
```

The first form of the `\overrides` statement says that the rule with tag *RuleLab1* overrides the rule with the rule tag *RuleLab2* regardless of what the heads of those rules are. The second form of `\overrides` says the following. Let ρ_1 be a rule with the tag *RuleLab1* and head *H1*, and ρ_2 be a rule with the tag *RuleLab2* and head *H2*. Assume that the variables in ρ_1, ρ_2, and the `\overrides` rule are standardized apart (i.e., are not shared, which can be always achieved by renaming). Then for any substitution θ such that $\theta(H1) = \theta(AtomForm1) = \theta(H2) = \theta(AtomForm2)$, the rule-instance $\theta(\rho_1)$ overrides the rule-instance $\theta(\rho_2)$.

The `\opposes` predicate specifies which literals in the rule heads should be considered to be in conflict, if derived simultaneously. As with the `\overrides` predicate, `\opposes` has two forms:

```
\opposes(AtomForm1,AtomForm2).
\opposes(RuleLab1,AtomForm1,RuleLab2,AtomForm2).
```

The first form of the `\opposes` predicate says that the base formulas *AtomForm1* and *AtomForm2* contradict each other. More precisely, for any variable substitution θ, the knowledge base must not infer $\theta(AtomForm1)$ if $\theta(AtomForm2)$ is also inferred, and vice versa. The 4-argument version of `\opposes` is more restrictive: it says that, for any substitution θ, the knowledge base must not infer $\theta(AtomForm1)$ by means of a rule with tag *RuleLab1*, if $\theta(AtomForm2)$ is inferred by means of a rule with tag *RuleLab2*, and vice versa. Note that by "inference" here we mean inference with respect to the argumentation theory in use—not just with respect to the usual first-order logic. For instance, `a :- b` and `b` do not necessarily imply `a` because the rule `a :- b` may be defeated by a different rule.

Both `\opposes` and `\overrides` can be defined via user rules as well as facts, and these rules and facts can even be added and deleted or disabled/enabled at run-time using the statements `insert{...}/delete{...}`, `insertrule{...}/deleterule{...}`, `enable{...}/disable{...}`. However, excessive use of these facilities is not recommended because the knowledge base might get hairy and hard to understand. This is especially true in the case of the `\opposes` predicate. Although it is advised to be conservative with these predicates lest the meaning of a knowledge base becomes obscure, there are no syntactic restrictions on the use of `\overrides` and `\opposes`: they can appear in rule bodies, heads, in aggregate functions, and so on.

It should be kept in mind that for any non-transactional literal L (base HiLog formula, frame, ISA, or subclass), L and `\neg` L always oppose each other, and there is no need to state this explicitly: it is part of the underlying theory of defeasible reasoning. Transactional literals do not participate in defeasible reasoning. They cannot be negated using `\neg`, but the compiler does not check if they appear as arguments to `\opposes`. If they do, this information is ignored.

38.4 Cancellation of Rules

Sometimes it is useful to turn off—or *cancel*—a rule instance, if certain conditions are satisfied. This can be specified using the special predicate `\cancel`. The default argumentation theory in FLORA-*2* understands two versions of that predicate:

```
\cancel(tag).
\cancel(tag,head).
```

The first rule cancels all rule instances whose tag unifies with *tag* and the second version cancels only those rules whose tag unifies with *tag* and whose head unifies with *head*.

Cancellation rules can also be tagged, overridden, or canceled by other cancellation rules. To illustrate, consider the following example:

```
t(aa).
t(bb).
@{L1} tt1(?X) :- t(?X).
@{L2} tt2(?X) :- t(?X).
@{L3} tt3(?X) :- t(?X).

@{c1} \cancel(?,tt1(bb)).
@{c2} \cancel(L2).
@{c3} \cancel(?,tt3(aa)).
@{c4} \cancel(?,tt3(bb)).
@{c5} \cancel(c4).
```

Here `tt1` is true only of `aa` and `tt3` only of `bb`. `tt2` is false for both `aa` and `bb`. For instance, `tt1(bb)` is false because of the cancellation rule `c1`. Similarly, `tt3(a)` is false because it is canceled by the rule `c3`. More interesting, however, is the reason for `tt3(bb)` being true. Note that rule tagged with `c4` cancels the derivation of `tt3(bb)`. However, rule `c5` cancels rule `c4`, so `tt3(bb)` stays put.

Observe the use of the 1-argument and 2-argument cancellation predicate. The first argument is always a term that matches rule tags for the rules to be canceled. The second argument, if present, matches the heads of the rules to be canceled. For instance, `\cancel(L2)` cancels only the rule tagged `L2`. If more than one rule matches `L2` (this is possible, but is not the case in our example), then all such rules are canceled. Two-argument cancellation predicates cancel only the rules that match both arguments. In our case, however, the first argument is a variable, as in `\cancel(?,tt3(aa))`, which means that all the rules matching the head are canceled, regardless of the tag.[19]

Here is a more complicated example:

```
device(printer).          abused(Bob,printer).       pardoned(printer,Bob).
device(scanner).          abused(Bob,scanner).       pardoned(scanner,Bill).
device(fax).              abused(Bill,scanner).
                          abused(Bill,printer).
                          abused(Mary,fax).
person(Bob), person(Bill), person(Mary).
@{id1}               authorized(?Persn,?Device) :- device(?Device), person(?Persn).
@{id2(?Dev,?Persn)}  \cancel(id1,authorized(?Persn,?Dev)) :- abused(?Persn,?Dev).
@{id3}               \cancel(id2(?Device,?Persn)) :- pardoned(?Device,?Persn).
```

The most interesting feature here is the tag `id2` parametrized with variables and the rule tagged

[19] Again, in our case, only one rule head matches the literal `tt3(a)`, but in general there can be several.

`id3`, which uses that parametrized tag in the head. Note also that here the cancellation rules are conditional. The effect is that rule `id3` cancels the instances of the cancellation rule tagged `id2(printer,Bob)` and `id2(scanner,Bill)`, which has the effect of authorizing Bob to use the printer and Bill to use the scanner, despite the reported abuses. The reader can verify that the net effect of all the cancellations and counter-cancellations is that Bill and Bob can use the fax, Bob and Mary are authorized to use the printer, and Bill and Mary can use the scanner.

Another interesting situation arises when cancellation rules are defeated not due to overriding, but due to a conflict with some other cancellation rule. For instance,

```
@{r}   P.
@{c1}  \cancel(r).
@{c2}  \neg \cancel(r).
```

Here rule `c1` cancels rule `r`, but `c1` conflicts with `c2`, since the two rules contradict each other. As a result, `P` stays true.

Finally, we should note that, if a rule that is being canceled overrides the canceling rule, than the first-mentioned rule stands and the cancellation rule is defeated instead. For example, in

```
@{L1} foo(1).
@{L2} foo(2).
@{L3} \cancel(?,foo(?)).
\overrides(L1,foo(1),?,\cancel(L1,foo(1))).
```

the instance `\cancel(L1,foo(1))` is defeated and `foo(1)` remains true. However, `foo(2)` gets canceled.

One might think that it makes little sense to specify the rule being canceled as having higher priority than the canceling rule and that it is simply a case of ill-design. However, there can be good reasons to design rules in this way. Suppose the following information is given at two different sites:

```
Site 1:
   @{L11} foo(1).
   @{L12} bar.

Site 2:
   @{L21} \cancel(?,foo(?)).
   @{L22} foo(2).
```

Site 1 might want to merge the rules from Site 2, but it is unwilling to let the `\cancel(?,foo(?))` statement tagged `L21` to have effect on the rules of Site 1. That is, for the merged set of rules, it

is ok to let the L22 statement to be canceled (because it was supposed to be canceled at Site 2), but it is not ok for the newly merged rules to cancel L21. To achieve this effect, we can add the following statements to the above:

```
origin(L11,site1).
origin(L12,site1).
origin(L21,site2).
origin(L22,site2).
\overrides(?lab1,?, ?lab2,?) :- origin(?lab1,site1), origin(?lab2,site2).
```

The first four facts simply tell us where each rule came from. The last rule says that the rules that came from site1 take precedence over the rules that came from site2. Note that we are in the same situation as described earlier: the fact L21 cancels both L11 and L22, but L11 has higher priority. As a result, L22 remains canceled and is false, but L11 is not canceled and remains true.

38.5 Changing the Default Defeasibility Status

In Section 36, we discussed the default defeasibility policy and the fact that defeasibility can be changed using the makestrict/makedefeasible primitives. This policy says that rules that have no explicit tags and no explicit defeasibility descriptor @@{defeasible} are considered strict. To change this default policy, *FLORA-2* provides two compiler directives:

```
:- default_is_defeasible_rules.
:- default_is_defeasible.       // short form
:- default_is_strict_rules.
:- default_is_strict.           // short form
```

The \default_is_defeasible_rules directive changes the default so that untagged rules become defeasible. The directive \default_is_strict_rules changes the default back to strict. These directives can appear any number of times in the file, changing the treatment of the untagged rules to the desired default. (Of course, such frequent switching is not advisable.)

Sometimes it is useful to be able to query the rule base based on rule tags and heads. Such queries can be issued using the clause{...} primitive. However, if you do not need to query the rule bodies, a faster fay is to use the tag primitive. For instance, if we have the rules

```
:- use_argumentation_theory.
abc(?X) :- cde(?X).
@{p} foo :- bar.
@@{defeasible} abc2(?X) :- cde2(?X).
@@{strict} foo2 :- bar2.
```

then the query

```
?- tag{?X,?Y@main}.
```

will return the following answers:

```
?X = 2
?Y = ${abc(?_h8660)@main}

?X = 4
?Y = ${abc2(?_h8623)@main}

?X = 5
?Y = ${foo2@main}

?X = p
?Y = ${foo@main}
```

Note that the query `tag{?X,?Y}` (without the module specification for the head, will likely return many answers, as there can be many rules in different loaded modules.

38.6 Supported Argumentation Theories

At present, $\mathcal{F}$LORA-*2* supports several different argumentation theories: the cautious, the original, and the strong courteous logics plus also a logic with general exclusion constraints. The $\mathcal{F}$LORA-*2* library of argumentation theories also contains a number of experimental packages that are not described here.

38.6.1 The Cautious, Original, and Strong Courteous Argumentation Theories

These argumentation theories are very similar to each other; the differences lie on the edges. All these theories use the `\opposes` and `\overrides` predicates as well as the notions of rebuttal, refutation, and cancellation as the means for determining which rule instances are to be defeated. The difference is that, in the original courteous theory, a rule, R, is defeated if another rule defeats, rebuts, or cancels R. In the cautious theory, that other (defeating) rule must not itself be defeated in order to defeat R. The cautious theory does not also let that other rule be involved in a circular defeating relationships, i.e., there can be no sequence of tagged rule instances (i_1, ρ_1), (i_2, ρ_2), ..., (i_{n-1}, ρ_{n-1}), (i_n, ρ_n), such that $i_n = i_1$, $\rho_n = \rho_1$, and at each step (i_i, ρ_i) either refutes or rebuts (i_{i+1}, ρ_{i+1}).

The strong Courteous argumentation theory is in-between the cautious and the original Courteous theories. It is very similar to the original theory except that it does not allow the defeating rule, ρ_1, to be rebutted or refuted by the rule ρ_2 that is being defeated by ρ_1.

The cautious argumentation theory is the default and should be good for most applications. It can be invoked by placing

```
:- use_argumentation_theory.
```

at the top of a $\mathcal{F}$LORA-*2* module where defeasible reasoning is to be used. This loads the new courteous argumentation theory into the builtin module `\gcl`. For the original courteous argumentation theory, place the following at the top:

```
:- use_argumentation_theory{ogcl}.
?- [ogclp>>ogcl].
```

The original argumentation theory is then available in the module `ogcl` (which stands for "original gcl;" of course, the user can choose a different module name).

The strong argumentation theory can be requested as follows (where, again, the choice of the module is up to the user):

```
:- use_argumentation_theory{sgcl}.
?- [sgclp>>sgcl].
```

38.6.2 Courteous Logic with Exclusion Constraints

This argumentation theory allows a group of facts larger than two to oppose to each other. This does not mean that the facts in that group are pairwise exclusive. Instead, it means that the facts in the group cannot be true together (but subsets of these facts can be true).

The argumentation theory with exclusion constraints can be invoked by placing the following at the top of an appropriate module:

```
:- use_argumentation_theory{gcle}.
?- [gclpe>>gcle].
```

Again, the user can choose to load this argumentation theory into a differently named module.

The main difference between this argumentation theory and the two previous ones is that it allows more than two rule heads to oppose each other. The syntax is

```
id:\Exclusion[\opposers->{opposer1, ..., opposerN}].
```

This kind of statements is called an *exclusion constraint* and it means that `opposer1`, ..., `opposerN` cannot be all true at the same time. The opposers `opposer1`, ..., `opposerN` must be all reified. The term `id` is the identifier of the exclusion constraint. The usual `\opposes` statements are also understood; they are treated as binary exclusion constraints. As with `\opposes`, the above exclusion constraints can have variables and they can also be defined by rules.

In case all opposers in an inclusion constraint are true, overriding determines which of them are defeated. Intuitively, the defeated opposers are those that do not "beat" (rebut) any other opposer in the same exclusion constraint.

38.7 Defeasible Rules Must Be Purely Logical

It must be kept in mind that all rules involved in defeasible reasoning must be *purely logical*. This includes both the tagged rules of the core knowledge base (whether they are explicitly mentioned in the overrides/opposes statements or not) as well as the rules that define the predicates `\overrides` and `\opposes`. If such a rule depends on a strict rule then the latter must also be purely logical. "Purely logical" here means that the bodies of such rules *cannot* use non-logical or dynamic features, such as:

- I/O statements
- Insert/delete statements
- The cut (!), the predicates `ground/1`, `var/1`, `nonvar/1`, and similar. These predicates can be used only if the alternatives (when the predicates fail) cause an abort and issue an error message for the user. For instance, the following predicate

  ```
  %check_state(?s) :- ground(?s), !.
  %check_state(?s) :- abort(['Nonground state found: ', ?s,
                             '. Might cause infinite recursion.'])@\sys.
  ```

  can be used in the body of a defeasible rule (or of the rules defining `\overrides` and `\opposes`) because the alternative to being ground here aborts the inference process. This is because `ground/1` is a logically clean predicate: if its argument is not ground, the predicate `%check_state(?s)` aborts.
- *Modal* predicates that require bound arguments, which include:
  - Comparison operators >, <, etc., unless it is assured that both sides of the comparison are ground during the inference.

- Inequality operators, such as `!=`, `!==`, `\=`, `\==`, unless it is certain that both arguments get bound during the evaluation.
- The evaluation operator *left* `\is` *right*, unless *right* is bound to an appropriate evaluable expression.

In all these cases, a general remedy in case of a runtime error that complains about unbound arguments is to identify the appropriate domain for the variables that must be bound and bind the variables in question before the modal predicate is invoked. For instance, suppose we wish to say that argument 2 of the predicate `price` must not have two different values for the same item (in argument 1):

```
\opposes(price(?x,?p1),price(?x,?p2)) :- ?p1 \= ?p2.
```

Unfortunately, such a definition will almost certainly cause a runtime error because one of the `?p1` or `?p2` will end up unbound in the course of the reasoning performed inside the argumentation theory. However, we do know that these variables must be bound to prices and thus we could write

```
\opposes(price(?x,?p1),price(?x,?p2)) :-
      price(?x,?p1),
      price(?x,?p2),
      ?p1 \= ?p2.
```

This will likely bind `?p1` and `?p2` before the inequality \= is used.[20] On the other hand,

```
\opposes(price(?x,?p1),price(?x,?p2)) :- ?p1 != ?p2.
```

is likely to work fine because `!=` is delayed until both arguments become bound. Generally, `!=` and `!==` are slower than but safer and more declarative than `\=` and `\==` because their evaluation is delayed when necessary and they are not that dependent on their position in the rule body. The comparison operators <, >, etc., as well as the `\is` operator are also generally safe, *if* their arguments get bound eventually.

[20] We say "likely" because `price(?x,?p)` itself may be defined by rules (rather than by a collection of facts) and these rules might involve comparisons, arithmetic operations, etc., and thus may require that it be called with some arguments (e.g., `?x`) bound. However, it cannot be guaranteed that, in the above rule, `price(?x,...)` will always be called with `?x` bound.

38.8 Debugging Defeasible Knowledge Bases

To help the user debug defeasible knowledge base, argumentation theories provide a special API, which can be used to find out why certain inferences were or were not made. The API consists of several methods, which take status-objects of the form `status(ruleTag,ruleHead)` and returns information such as why a certain rule was defeated, which rules are defeated by the given one, etc. The list of methods follows:

- `status(?T,?H)[howDefeated -> ?Reason]`.
  Here `?T` is a rule Id and `?H` is a rule head. The rule head cannot be a variable. There is no need to reify the rule head: $\mathcal{F}$LORA-*2* understands that a rule-head literal is expected and will compile it accordingly.

  If the corresponding rule is not defeated, the query fails. Otherwise, `?Reason` is the result of the query. It can take three different forms:

  - `refutedBy(ruleTag,ruleHead)`: In this case, all rule tag/head pairs that refute the rule(s) represented by the `?T/?H` pair (i.e., whose tag unifies with `?T` and head with `?H`) will be returned.
  - `rebuttedBy(ruleTag,ruleHead)`: All the rule tag/head pairs that rebut the rule(s) represented by the `?T/?H` pair will be returned.
  - `disqualified`: This is returned if the rule is disqualified. In the default `\gcl` theory, a rule is disqualified if it is canceled, overridden by a strict rule, or if it transitively refutes/rebuts itself. In this case, an auxiliary method, `howDisqualified`, can provide additional information, as described next.
  - `canceled`: Other argumentation theories disqualify a rule only if it is canceled. These theories return `canceled` in this case instead of `disqualified`.
  - `beatenByStrictRule(?ruleHead)`: This means that a strict rule with the head `?ruleHead` opposes the rule with tag `?T` and head `?H`.

- `status(?T,?H)[howDisqualified->defeatCycle(?Defeater,?Defeated)]`.
  If the rules represented by the `?T/?H` pair are disqualified (in the default argumentation theory), this method returns the set of terms of the form `defeats(?Defeater,?Defeated)`. Here both `?Defeater` and `?Defeated` are tag/head pairs that are defeated by the `?T/?H` pair and, in addition, `?Defeater` defeats `?Defeated`. By following these pairs one should be able to discover a cycle of defeats starting and ending with `?T/?H`, which constitutes a self-defeating cycle.

- `status(?T,?H)[howDisqualified->canceled]`.
  This query is true if all the rules whose tag unifies with `?T` and head with `?H` are canceled.

- `status(?T,?H)[howDisqualified->beatenByStrictRule(?SRH)]`.
  This literal is true if all the defeasible rules with tag `?T` and head `?H` have an opposing strict rule. In that case, `?SRH` is bound to the head of that opposing rule.
- `status(?T,?H)[info->?Info]`.
  This method provides all kinds of details about the behavior of the rule with head `?H` and tag `?T`. The information returned includes `candidate` (if the corresponding rule is a candidate), `conflictsWith(?Head)`, `competes(?Exclusion,?Head)`, `refutes(?Head)`, and `rebuts(?Head)`.

All these methods are provided by all argumentation theories and are available in their respective modules (`\gcl` for the default argumentation theory; for other argumentation theories, these methods are available in the modules in which these argumentation theories are loaded). However, the information returned by these methods differs from one argumentation theory to another. For instance, for the default, `sgclp`, and `ogclp` theories, the method `howDefeated` may return `refutedBy(...)` and `rebuttedBy(...)`, but in case of `gclpe` this method might return `notBeaterFor(ExclusionConstraintId)` as well as `canceled` and `beatenByStrictRule(?SRH)`. Here are some of the examples of these queries:

```
?- status(?T,configuration(0,block4,square7))[info->conflictsWith(?X)]@\gcl.
?- status(?T,configuration(nxt(nxt(0)),block4,square3))[howDefeated->?X]@\gcl.
?- status(?T,configuration(0,block4,square7))[info->competes(?Exclusion,?O)]@gcle.
```

39 Primitive Data Types

$\mathcal{F}$LORA-*2* supports the following built-in data types: `\boolean`, `\long`, `\integer`, `\double`, `\decimal`, `\string`, `\symbol`, `\charlist`, `\iri` (international resource identifier; generalization of URLs), `\time`, `\date`, `\dateTime`, `\duration`, and `\currency`.

Following the now accepted practice on the Semantic Web, $\mathcal{F}$LORA-*2* denotes the constants that belong to a particular primitive data type using the idiom `"`*`literal`*`"^^`*`type`*. The *literal* part represents the *lexical form* of the constant and the *type* part is the type name. For instance, `"2004-12-24"^^\date`, `"2004-12-24T15:33:44"^^\dateTime`. For most of the data types, specific syntax is expected in the lexical part of the constant or a parsing error will be issued.

A type name must be a Prolog 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 URL. For instance, `'http://www.w3.org/2001/XMLSchema#time'`. However, for convenience, all type names will have one or more $\mathcal{F}$LORA-*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 39.2 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 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 type `\time`. This mechanism is more general and allows bounding of variables to arbitrary classes, not just data types; it has already been discussed in Section 11.

Note: "primitive" does not mean "atomic" and most of the data types have non-trivial internal structure. In other ways, a primitive data type constant is an object and it should not be "hacked": the components of these constants should be accessed only via methods described in this section.

The methods that are applicable to each particular primitive type vary from type to type. However, certain methods are more or less common:

- `toSymbol`, which applies to a data type constant and returns its printable representation (a Prolog atom). For instance, if ?Y is bound to `"12:44:23"^^\time` then `?Y[toSymbol-> '12:44:23']@\basetype` will be true. Similarly, `"http://foo.com/bar"^^\iri[toSymbol-> 'http://foo.com/bar']@\basetype` is true. Note that `"http://foo.com/bar"^^\iri` and `'http://foo.com/bar'` are very different objects! The first is an IRI-object and the second a symbol-object.

- `toType`(parameters), which applies to 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]@\basetype`. The other will apply to the general constant symbol (≡ Prolog atom) representation of the data type. For instance, `\time[toType('12:23:45')-> "12:23:45"^^\time]@\basetype`.

- `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). For instance, `\time[isTypeOf("12:13:44"^^\time)]@\basetype` will be true, while `\time[isTypeOf(123)]@\basetype` false.

- `equals`(constant), which tells when the given datatype constant equals some other term. For instance, the following should be true: `"2018-12-23T0:0:0"^^\dateTime[equals("2018-12-23"^^\dateTime)]@\basetype`.

- `lessThan`(constant) or `lessEq`(constant), which tells when one constant is less than 9respectively, less or equal) some other term. 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. Example:
  `"2018-12-23T0:0:0"^^\dateTime[lessThan("2018-12-23T1:0:0"^^\dateTime)]@\basetype.`
  `"2018-12-23T0:0:0"^^\dateTime[lessEq("2018-12-23T1:0:0"^^\dateTime)]@\basetype.`
- `typeName`, which tells the type name (and thus also class) of the given data type. For instance, the query `123[typeName->?N]@\basetype` binds `?N` to `\long`, and the query `"abc"^^\string[typeName->?N]@\basetype` to `\string`.

All these methods are available in the *FLORA-2* system module `\basetype`.

In addition, each primitive data type has a built-in 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 with the same name.

Note: Since built-in classes have infinite extensions, *FLORA-2* allows only ground membership tests with respect to these classes. Non-ground tests are permitted, but evaluate to **false**, `true`, or *undefined* depending on the situation. For instance, the following queries have the specified results:

```
?- abc:\symbol.       // true
?- f(?X):\integer.    // false
?- ?X:\symbol.        // undefined
?- ?X:\integer, ?X=1.    // true
?- ?X:\integer, ?X=abc.  // false
```

If at the end of query evaluation the variable `?X` in a subgoal like `?X:\integer` remains unbound, the subgoal evaluates to `undefined`, which leaves `?X` unbound. □

The following subsections describe each data type separately and in greater detail.

39.1 The *FLORA-2* \symbol Data Type

Before describing the actual data types, we remind that in Section 8.1 we introduced alphanumeric constants, such as `abc12`, and sequences of characters enclosed in single quotes, such as `'aaa 2*)@'`, and called them *general constant symbols* or *Prolog atoms*. These are not the only constants in *FLORA-2*. In the following subsections we will introduce typed literals that represent time, date, and more.

The general constant symbols mentioned above are partitioned into three disjoint subcategories: *strings* (class `\string`) , *IRIs* (class `\iri`), and *abstract symbols* (class `\symbol`).

It is important to *not* confuse the `\symbol` data type with the general notion of a symbol, like "constant symbol," "variable symbol," or "function symbol" encountered in previous sections. The abstract symbols that are members of the `\symbol` data type are all the Prolog atoms that are disjoint from constants of type `\iri` and `\string`. In other words, an abstract symbol is any Prolog atom that does not serve as internal representations for the data types of the form `"..."^^\iri` or `"..."^^\strings`.[21]

A `\symbol`-object like `"abc&* 65"^^\symbol` is the same as the Prolog atom `'abc&* 65'`, but this is not true for IRIs and strings. For instance, `"abc&* 65"^^\string` is different from `'abc&* 65'`. It is represented by another Prolog atom but one should never attempt to manipulate the internal representation of `\string`- or `\iri`-objects except through the methods provided for them.

The class `\symbol` has only the following methods apart from the already mentioned `toSymbol`, `lessThan`, etc.:

- `\symbol[|toNumber => \number|]`
  When applied to a symbol that is convertible to a number, returns the result of that conversion:

  ```
  ?- '123.5'[toNumber->?R]@\basetype.
  ?P = 123.5000

  ?- '456'[toNumber->?R]@\basetype.
  ?P = 456
  ```

  When applied to a symbol that is not convertible to a number, this method issues a warning and returns the input object.
  When applied to a number, it returns that number (without a warning). In all other cases, this method returns a "No."

- `\symbol[concat(`*`List`*`) => \object]`
  Note that that this is *not* a class-level signature, but an object level one. That is, it applies to the object `\symbol` itself and not to the individual objects that belong to the class `\symbol`. For instance,

  ```
  flora2 ?-  \symbol[concat([abc,cde,fgh])->?X]@\basetype.

  ?X = abccdefgh
  ```

[21] For efficiency, IRIs and strings are encoded as atoms that are prefixed with a character sequence that is unlikely to be engaged by the user. IRIs are thus atoms prefixed with the character `i` and the backspace character; strings are atoms that are prefixed with the character `s` and the backspace symbol.

The *List* argument in the `concat` method can be a list of anything, not necessarily of other symbols. For instance,

```
flora2 ?-  \symbol[concat(["11:11:11"^^\time,cde,fgh])->?X]@\basetype.

?X = '11:11:11cdefgh'
```

- `\symbol[|=> contains(\symbol` or `\string)|]`
  Tells if the second symbol (or `\string`) is contained in the first.
- `\symbol[|contains(\symbol` or `\string) => \list|]`
  Tells if the second symbol (or `\string`) is contained in the first. The result is returns as a list of two numbers: the position of the beginning of the match and of the end of the match. Positions start with 1.
- `\symbol[|reverse => \symbol|]`
- `\symbol[|length => \integer|]`
- `\symbol[|toUpper => \symbol|]`
- `\symbol[|toLower => \symbol|]`
- `\symbol[|toList => \list|]`
  For instance, `abcde[toList->?P]@\basetype` will bind `?P` to `"abcde"^^\charlist`. Equivalently, one could write `\list[toType(abcde)->?P]@\basetype`.
- `\symbol[|=> startsWith(\symbol` or `\string))|]`
- `\symbol[|=> endsWith(\symbol` or `\string))|]`
- `\symbol[|subsymbol(\integer,\integer) => \symbol|]`
  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.

39.2 The \iri Data Type

The canonical representation of constants of type IRI (international resource identifiers, a generalization of URLs, uniform resource locators) is

```
"some iri"^^\iri
```

where *literal* must have a lexical form corresponding to IRIs on the World Wide Web. IRIs have an *abbreviated* syntax also:

```
\"some iri"
```

The full name of the IRI type is `'http://www.w3.org/2007/rif#iri'`.

IRIs can come in the usual expanded syntax (full or abbreviated) or in a compact form known as the *curi* form (for *compact IRI*).

A *compact form* of an IRI (curi) 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 an alphanumeric identifier or a quoted sequence of characters. If *LOCALNAME* contains non-alphanumeric symbols, it must be enclosed in quotes (single or double), as in `"ab%20"` or `'ab%20'`). A compact IRI is treated as a macro that expands into a full IRI by concatenating the expansion of *PREFIXNAME* with LOCALNAME (sans the external quotes). That is, if the expansion of a prefix, `pref`, is `http://foo.bar.com/` then `pref#'abc/cde'` (and `pref#"abc/cde"`) expands into `"http://foo.bar.com/abc/cde"^^\iri`. Note that, in a curi, `#` is an infix operator and the left and right sides are its arguments. In this example, the argument `abc/cde` has a non-alphanumeric character, `/`, so the argument is quoted (using single quotes) so it would be a proper symbolic constant.

The prefix of a compact IRI must be defined in one of the following ways:

```
:- iriprefix{PREFIXNAME = PREFIXIRI}.
:- irilocalprefix{PREFIXNAME = PREFIXIRI}.
?- iriprefix{PREFIXNAME = PREFIXIRI}.
```

Here `PREFIXIRI` can be an alphanumeric identifier or a quoted atom (the quotes can be single or double). Prefixes can also be defined at run time using a query:

```
?- 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"^^\iri`. Likewise,

```
?- ?Y = AAAWEB#"ab%20"
```

binds `?Y` to `"http://www.AAA.com/ab%20"^^\iri`.

39.2.1 Scope of IRI Prefixes

Module locality. IRI prefix definitions are local to the module where they are defined. If we define the following in module `foo`:

```
:- iriprefix{W3='http://w3.org/'.
C[a->"http://w3.org/abc"^^\iri].
```

and then load the following file into module `main`

```
r(?X):-?X[a->W3#abc]@foo.
s(?X):-?X[a->W3#cde].
```

then `W3` will have an expansions for calls to `foo`, but not to the current module. Thus, the answer to

```
?- r(?X).
```

will be `C` but

```
?- s(?X).
```

will get an error saying that the prefix W3 is not defined. If the same prefix is defined both in the file and in a module being referenced by a query then the prefix definition in the current file takes precedence. For instance, if in the above example the module main had another declaration for the IRI prefix `W3` then `?X[a->W3#abc]@foo` will use that definition rather than the one in module `foo`.[22]

The prefix macro-expansion works also for transactional predicate and method names. For instance,

[22] The rationale here is the theory of the "least surprise."

```
:- iriprefix{W3 = 'http://w3.org/'}.
C[%W3#aaa(b)].
%W3#r(?Y)(?X):-?X[%W3#abc(?Y)]@foo.
```

Note, however, that transactional (%-prefixed) symbols can occur only as the names of predicates and methods (possibly higher-order predicates and methods).

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#'
swrlb 'http://www.w3.org/2003/11/swrlb#').
dc    'http://purl.org/dc/elements/1.1/').
```

However, one can always override these built-in definitions using either a compile time directive `iriprefix` or a runtime query `iriprefix`.

File locality. The `iriprefix` directives are *global* within the module. This means that these directives take effect not only in the file in which they appear, but also in all files *added* to the same module afterwards. Prefixes defined in the files loaded/added to module `main` also become known in the $\mathcal{F}$LORA-*2* shell. However, if an *added* file has an `iriprefix` directive for a previously defined prefix then the new definition overrides the old one (and is inherited by all subsequently added files). Note: *loading* a file into a module zaps all the prefix definitions known to the module and replaces them with whatever is defined in that file.

Sometimes, however, it is preferable to have prefix definitions that are truly local to a file and are not inherited by subsequently added files. This can be done using the `irilocalprefix` directive. For instance, suppose file A was loaded into module `Mod` and has these prefix definitions:

```
:- iriprefix{foo='http://123'}.
:- iriprefix{moo='http://abc'}.
:- iriprefix{doo='http://456'}.
:- irilocalprefix{bar='http://cde'}.
```

Suppose also that file B has these definitions and was added to `Mod` later:

```
:- iriprefix{foo='http://789'}.
:- irilocalprefix{moo='http://fgh'}.
```

Finally, suppose file C was added later to the same module `Mod`. Then file A has prefixes `foo`, `moo`, `doo`, and `bar`. File B does not have prefix `bar` defined because it is local to file A. Moreover, prefixes `foo` and `moo` are overridden and have a different meaning than they have in file A. Prefix `doo` has the same meaning, however.

In file C, the situation is as follows. Prefixes `foo`, `moo`, `doo` are inherited. Prefix `doo` has the same meaning as in files A and B. However, prefix `foo` was overridden in file B, so it is file B's expansion that is inherited by file C. As to prefix `moo`, it is inherited from file A and has the same meaning. Note that even though `moo` was overridden by file B, this overriding was local and did not propagate to file C.

Note that

```
:- iriprefix{foo=bar}.
```

is equivalent to the pair

```
:- irilocalprefix{foo=bar}.
?- iriprefix{foo=bar}.
```

The `irilocalprefix` directive takes case of compiling prefix `foo` within the given file and the executable `iriprefix` directive propagates the prefix to all subsequently added files (and to the *F*LORA-*2* shell, if applicable).

39.2.2 IRI Prefix Querying and Decomposition

*F*LORA-*2* also provides the necessary primitives to enable querying the available prefix definitions at run time: `prefix{?prefix,?expansion}` and `prefix{?prefix,?expansion}@module`. When `@module` is not specified, the current module is assumed. When `module` is given, only the prefixes defined for the given module are returned. The aforesaid predefined prefixes are considered to be defined for every module. For example:

```
:- iriprefix{foo = 'bar.com'}.
?- prefix{foo,?exp}@?M.

?exp = bar.com
?M = main


?- prefix{?p,?exp}@main.
```

```
?p = foo
?exp = bar.com

?p = owl
?exp = http://www.w3.org/2002/07/owl#

?p = rdf
?exp = http://www.w3.org/1999/02/22-rdf-syntax-ns#

?p = rdfs
?exp = http://www.w3.org/2000/01/rdf-schema#

?p = rif
?exp = http://www.w3.org/2007/rif#

?p = xsd
?exp = http://www.w3.org/2001/XMLSchema#
```

In addition, given an IRI, one can decompose it into the prefix and the local name. If different prefixes match, all are returned via backtracking. The query also returns the modules in which those prefixes are defined:

```
irisplit{IRI,Prefix,LocalName,Module}
```

If the first argument is not an IRI, this primitive returns `\false`.

39.2.3 Class \iri

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 the $\mathcal{F}$LORA-*2* module `\basetype` (or, in the abbreviated form, `\btp`). 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 IRI scheme. For `http`, `ftp`, `file`, etc., the meaning of 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 IRI schemes. For instance, for the `urn` and `file` schemata, only the path component is defined. For the `mailto` scheme, port, path, query, and fragment are not defined. If a scheme is not recognized, then the part of the IRI that follows the scheme goes into the `path` component unparsed.

Other methods:

- `\iri[|toSymbol => \symbol|]`
- `\iri[|=> equals(\object)|]`
- `\iri[|typeName => \symbol|]`

Examples:

- `"http://foo.bar.com/abc"^^\iri` — the full syntax
- `\"http://foo.bar.com/abc"` — the abbreviated syntax
- ```
  ?- \iri[toType('http://foo.bar.com/abc') ->
                              "http://foo.bar.com/abc"^^\iri]@\basetype
  ```
- `?- "http://foo.bar.com/abc"^^\iri[host -> 'foo.bar.com']@\btp`
- Same query using the abbreviated syntax:
  `?- \"http://foo.bar.com/abc"[host -> 'foo.bar.com']@\btp`
  ```

39.3 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 (1 or -1), year, month, day, hour, minute, second, zone sign (1 or -1), zone hour, zone minute. All arguments, except date sign and zone sign, are assumed to be positive integers; date sign and zone sign can be either 1 or -1.
- `\dateTime[toType(\symbol) => \dateTime]`
  Here the argument of `toType()` is expected to be a symbol (atom) of the form that is allowed inside the quotes in `"..."^^dateTime`.
- `\dateTime[=> isTypeOf(\object)]`
  Tells if object belongs to the primitive type `\dateTime`.
- `\dateTime[now => \dateTime]`
  Current local date+time.
- `\dateTime[now(utc) => \dateTime]`
  Current UTC date+time.
- `\dateTime[now(\decimal) => \dateTime]`
  Current UTC date+time adjusted for the time zone given by the argument. The decimal argument has the form `SHH.MM`, where `S` is the plus sign (or empty) or the minus sign. If more than two `MM` digits are provided, the rest are cut off. If `HH>24` then the query fails.

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|]`
- `\dateTime[|date => \date|]`
- `\dateTime[|time => \time|]`

Note: the methods `date` and `time` listed above exist only in $\mathcal{ERGO}$.

Other methods:

- `\dateTime[|toSymbol => \symbol|]`
- `\dateTime[|=> equals(\object)|]`
- `\dateTime[|=> lessThan(\object)|]`
- `\dateTime[|=> lessEq(\object)|]`
- `\dateTime[|typeName => \symbol|]`
- `\dateTime[|add(\duration) => \dateTime|]`
  Note: any two of the arguments must be bound, which means that this method can perform subtraction.
- `\dateTime[|minus(\dateTime) => \duration|]`

Note: the method `minus` exists only in $\mathcal{ERGO}$.

Examples:

- `\dateTime[toType('2011-10-10T12:33:55.23')->?DT]@\basetype` binds `?DT` to `"2011-10-10T12:33:55.23"^^\dateTime`
- `"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[toSymbol -> '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]@\btp.`
- ```
  ?- "2011-12-22+2:19"^^\dt[minus("2019-11-29T2:30:30-2:09"^^\dt)->
                            "-P0007Y11M07DT02H30M30S"^^\duration]@\btp
  ```

Note: the example involving `minus` will not work in $\mathcal{F}$LORA-*2*— only in $\mathcal{ERGO}$.

39.4 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 (1 or -1), year, month, day, zone sign (1 or -1), zone hour, zone minute. All arguments, except date sign and zone sign, are assumed to be positive integers; 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`.
- `\date[now => \date]`
  Current local date.
- `\date[now(utc) => \date]`
  Current UTC date.
- `\date[now(\decimal) => \date]`
  Current UTC date adjusted for the time zone given by the argument. The decimal argument has the form `SHH.MM`, where `S` is the plus sign (or empty) or the minus sign. If more than two `MM` digits are provided, the rest are cut off. If `HH>24` then the query fails.

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[|toSymbol => \symbol|]`
- `\date[|=> equals(\object)|]`
- `\date[|=> lessThan(\object)|]`
- `\date[|=> lessEq(\object)|]`
- `\date[|typeName => \symbol|]`
- `\date[|add(\duration) => \date|]`
  Note: any two of the arguments must be bound, which means that this method can perform subtraction.
- `\date[|minus(\date) => \duration|]`
- `\date[|toDateTime(\integer,\integer,\decimal) => \dateTime|]`
  The arguments are hours, minutes, and seconds (with possible milliseconds).

Note: the methods `toDateTime` and `minus` exist only in $\mathcal{ERGO}$.

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[toSymbol -> '2001-11-23+00:00']@\basetype.`
- ```
  ?- "2011-12-22"^^\date[toDateTime(11,15,7.6) ->
                            "2011-12-22T11:15:07.6"^^\dateTime]@\basetype.
  ```
  This example will not work in $\mathcal{F}$LORA-*2*— only in $\mathcal{ERGO}$.
  ```
  ?- "2011-12-22+2:09"^^\d[minus("2019-10-29+3:19"^^\d)->
                            "-P0007Y10M07DT00H00M00S"^^\du]@\btp.
  ```
  This example will not work in $\mathcal{F}$LORA-*2*— only in $\mathcal{ERGO}$.
  ```
  ?- "2001-11-23-02:30"^^\date[add("-P2Y2M10DT"^^\duration) ->
                                    "1999-09-13-02:30"^^\date]@\basetype.
  ```
  Note that when adding a duration to a date, the time-part of the duration constant must be empty.
  ```

39.5 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 this 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`
- `\time[now => \time]`
  Current local time.
- `\time[now(utc) => \time]`
  Current UTC time.
- `\time[now(\decimal) => \time]`
  Current UTC time adjusted for the time zone given by the argument. The decimal argument has the form `SHH.MM`, where `S` is the plus sign (or empty) or the minus sign. If more than two `MM` digits are provided, the rest are cut off. If `HH>24` then the query fails.

Component methods:

- `\time[|hour => \integer|]`

- `\time[|minute => \integer|]`
- `\time[|second => \integer|]`
- `\time[|zoneSign => \integer|]`
- `\time[|zoneHour => \integer|]`
- `\time[|zoneMinute => \integer|]`
  The arguments are years, months, and days.

Other methods:

- `\time[|toSymbol => \symbol|]`
- `\time[|=> equals(\object)|]`
- `\time[|=> lessThan(\object)|]`
- `\time[|=> lessEq(\object)|]`
- `\time[|typeName => \symbol|]`
- `\time[|add(\duration) => \time|]`
  Note: any two of the arguments must be bound, which means that this method can perform subtraction.
- `\time[|minus(\time) => \duration|]`
- `\time[|toDateTime(\integer,\integer,\integer) => \dateTime|]`

Note: the methods `minus` and `toDateTime` exist only in $\mathcal{ERGO}$.

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[toSymbol -> '12:44:55']@\basetype.`

- `?- "20:12:22"^^\time[toDateTime(2011,12,22) ->`
  `"2011-12-22T20:12:22"^^\dateTime]@\basetype.`
  This example will not work in $\mathcal{F}$LORA-*2*— only in $\mathcal{ERGO}$.
  `?- "11:22:33+2:22"^^\t[minus("12:23:44+2:09"^^\t)->`
  `"-P0000Y00M00DT01H01M11S"^^\du]@\btp.`
  This example will not work in $\mathcal{F}$LORA-*2*— only in $\mathcal{ERGO}$.
  `?- "12:44:55"^^\time[add("PT2M3S"^^\duration) -> "12:46:58"^^\time]@\btp.`
  Note that when adding a duration to a time constant, the date-part of the duration constant must *not* be present.

39.6 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 `"sPnYnMnDTnHnMdS"^^\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, day, 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 integer (for instance, the number of hours can be more than 12 and the number of minutes and seconds can exceed 60) and `d` stands for a decimal number. 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,`
  `\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 an 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[|toSymbol => \symbol|]`
- `\duration[|=> equals(\object)|]`
- `\duration[|=> lessThan(\object)|]`
- `\duration[|=> lessEq(\object)|]`
- `\duration[|typeName => \symbol|]`
- `\duration[|add(\duration) => \duration|]`
  Note: any two of the three arguments must be bound. This means that this method can also do subtraction. However, it should be noted that subtraction of positive durations is not always well-defined because the result depends on the context (leap vs. non-leap years). So, in some such cases this method might issue an error.
- `\duration[|inverse => \duration|]`
  This changes the sign of a duration.

Examples:

- `"P5Y5M10DT11H24M22S"^^\duration`
- `?- "-P2Y05M10DT11H24M22S"^^\duration[minute -> 24]@\basetype.`
- ```
  ?- "-P2Y05M10DT11H24M22S"^^\duration[inverse ->
          "P2Y05M10DT11H24M22S"^^\duration]@\basetype.
  ```
  ```

39.7 The Primitive Type \currency

The primitive type for currency represents the different currencies in the world. A currency constant has the form "*xyz amount*"^^\currency, where *xyz* is the three-letter international symbol for the currency and *amount* is a number. For instance, "USD 12.4"^^\currency, "GBP 21.4"^^\currency, "EUR 14"^^\currency. Spaces and commas are removed before the currency literal is parsed, and the currency symbol is capitalized. The currency symbol can also appear after the amount. For instance, " 12,34 5.89 ILS "^^\currency gets standardized by the parser to "ILS 12345.89"^^\currency.

Class methods:

- \currency[toType(\symbol,\decimal) => \currency]
  \currency[toType(\decimal,\symbol) => \currency]
  The symbolic argument is expected to be a valid currency 3-letter code. The numeric argument is the amount.
- \currency[toType(\symbol) => \currency]
  Like the above but expects a single symbolic argument like 'EUR 123' or '234 KGS'.
- \currency[=> isTypeOf(\object)]
  Tells if an object belongs to the primitive type \currency.

Component methods:

- \currency[|unit => \symbol|]
  The unit of the currency (e.g., USD, ZAR, JPY).
- \currency[|amount => \decimal|]
  The amount specified.
- \currency[|description => \symbol|]
  Description of the currency (e.g., Indian Rupiah).
- \currency[|sign => \symbol|]
  The sign commonly used for the currency (e.g., '$'). Note: the same symbol is commonly used by many currencies. The $ sign is one of the most popular; it is used by over 20 currencies. Some currencies do not have a special sign, in which case null is returned.
- \currency[|add(\currency) => \currency|]
  \currency[|add(\number) => \currency|]

Adds a currency constant or a number to another currency constant. If adding two currency constants, they must have the same currency unit.

- `\currency[|times(\number) => \currency|]`
  Multiplies currency by a number.
- `\currency[|ratio(\currency) => \number|]`
  Divides one currency constant by another currency constant; returns a number. The two currency constants must have the same currency unit.

Other methods:

- `\currency[|toSymbol => \symbol|]`
- `\currency[|typeName => \symbol|]`
- `\currency[|lessThan(\currency)|]`
- `\currency[|lessEq(\currency)|]`

These methods work the same way as for other data types.

Examples: The following queries are all true:

- `?- "usd 123"^^\currency[unit -> USD]@\basetype.`
- `?- "33 Eur"^^\currency[sign -> 'âĆň']@\basetype.`
- `?- "33 GBP"^^\currency[amount -> 33]@\basetype.`
- `?- "ILS 36 "^^\currency[description -> 'New Israeli Sheqel']@\basetype.`
- `?- \currency[toType(123,GBP) -> "GBP 123"^^\currency]@\basetype.`
- `?- \currency[toType(' Ils 123') -> "ILS 123"^^\currency]@\basetype.`

39.8 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 the `\boolean` 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[toSymbol => \symbol]`
- `\boolean[=> isTypeOf(\object)]`

Other methods:

- `\boolean[|toSymbol => \symbol|]`
- `\boolean[|=> equals(\object)|]`
- `\boolean[|=> lessThan(\object)|]`
  Note: `\false[lessThan(\true)]`.
- `\boolean[|=> lessEq(\object)|]`
- `\boolean[|typeName => \symbol|]`
- `\boolean[|rawValue => \symbol|]`
  Extract the content value from the `\boolean` data type. For instance,
  `?- "true"^^\boolean[rawValue->?X]@\basetype.`
  `?X = true`

39.9 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.

In $\mathcal{F}$LORA-*2*, the `\float` and `\double` type designators are interchangeable, and the constants of these data types are treated as regular floating point numbers. This means that, for example, `"1.2"^^\double`, `"1.2"^^\float`, and `1.2` represent the same number.

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[toType(\string) => \double]`
  `\double[toType(\symbol) => \double]`
  Converts strings and symbols into doubles, if the textual representation of these values is a number.
- `\double[=> isTypeOf(\object)]`

Instance methods:

- `\double[|floor => \integer|]`
- `\double[|ceiling => \integer|]`
- `\double[|round => \integer|]`

Other methods:

- `\double[|toSymbol => \symbol|]`
- `\double[|=> equals(\object)|]`
- `\double[|=> lessThan(\object)|]`
- `\double[|=> lessEq(\object)|]`
- `\double[|typeName => \symbol|]`
- `\double[|rawValue => \double|]`
  Extract the number part of the `\double` data type.

Examples: `"2.50"^^\double`, `2.50`, `25E-1` — different forms of `\double`.
`?- "3.54"\double[round->?X]@\basetype.` // answer: `?X = 4`,
`?- 5.51[floor->?X]@\basetype.` //answer: `?X = 5`,
`?- \double[toType(51)->?X]@\basetype.` // `?X = 51.0000` — long-to-double conversion.

39.10 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]`
  `\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.
- `\long[=> isTypeOf(\object)]`

Other methods:

- `\long[|toSymbol => \symbol|]`
- `\long[|=> equals(\object)|]`
- `\long[|=> lessThan(\object)|]`
- `\long[|=> lessEq(\object)|]`
- `\long[|typeName => \symbol|]`
- `\long[|rawValue => \long|]`
  Extract the number part of the `\long` data type.

Examples: `123`, `55`, `"55"^^\long`.

39.11 The Primitive Types \decimal, \number, \integer, and \short

At present, $\mathcal{F}$LORA-*2* does not implement the `\decimal` and the `\integer` types, which correspond to XML Schema arbitrary precision types decimal and integer. Instead, `\decimal` and `\number` are synonyms for `\double`, while `\short` and `\integer` for `\long`, which are described in previous subsections. As usual, there are corresponding classes `\integer`, `\short`, `\number`, and `\decimal`.

39.12 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.

The following methods are available in module `\basetype`:

Class methods:

- `\string[=> isTypeOf(\object)]`
- `\string[toType(\object) => \string]`

Note that the method `toType` in class `\string` can be used to serialize any term as a string. For instance,

```
flora2 ?-  \string[toType(abc(cde))->?val]@\basetype.

?val = "abc(cde)"^^\string
```

Instance methods:

- `\string[|=> contains(\string)|]`
  Tells if the second string (or symbol) is contained in the first.
- `\string[|contains(\string` or `\symbol)) => \list|]`
  Tells if the second string (or symbol) is contained in the first. The result is returns as a list of two numbers: the position of the beginning of the match and of the end of the match. Positions start with 1.
- `\string[|concat(\string` or `\symbol) => \string|]`
- `\string[|reverse => \string|]`
- `\string[|length => \integer|]`
- `\string[|toUpper => \string|]`
- `\string[|toLower => \string|]`

- `\string[|toList => \string|]`
  For instance, `"abcde"^^\string[toList->?P]@\basetype` will bind `?P` to `"abcde"^^\charlist`. Equivalently, one could write `\list[toType("abcde"^^\string)->?P]@\basetype`.
- `\string[|=> startsWith(\string` or `\symbol)|]`
- `\string[|=> endsWith(\string` or `\symbol)|]`
- `\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[|=> lessEq(\object)|]`
- `\string[|typeName => \symbol|]`

Examples:

- `"abc"^^\string`
- `"a string\n"^^\string`
- `"a\tstring\b"^^\string`
- `"string with a 'quoted' substring"^^\string`
- `?- "abc"^^\string[concat("bbb"^^\string)->?X]@\basetype.`
  `?X = "abcbbb"^^\string`

Note that internally the string `"abc"^^\string` and the atom `'abc'` are different. To extract the actual pure Prolog atom (stripped from the internal stuff), use the method `rawValue`. For instance, `?- "abc"^^\string[rawValue->?X]@\basetype`.

39.13 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.
  Since checking list containment is a very common operation, this method has a special shortcut `\sublist`:

```
?- [b,?A] \sublist [b,c,a,d].
?A = c
?A = a
?A = d
[1,2,4] \sublist [1,2,3,4].
Yes
```

- `\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.
  Since checking list membership is a very common operation, this method has a special shortcut `\in`:

```
?- a \in [b,c,a,d].
Yes
```

  The same operator can also be used to check for numeric and other ranges:

```
?- 5.1 \in 2.1..6.7.
```

Note that the range expression 2.1 .. 6.7 is basically a list [2.1, 3.1, 4.1, 5.1, 6.1], so 5.1 is in it, but, say, 5.2 is not. To see what kind of a list is represented by a range expression, leave the left-hand side of the `\in` operator unbound:

```
?- ?X \in 3..5.
?X = 3
?X = 4
?X = 5
```

See Section 16 for more.

- `\list[|select(\object) => \list|]`
  Find an member in list that unifies with the object-argument and return the list with the selected member removed. For instance,

```
?- "[a,b(1),c,b(2)]"^^\list[select(b(?X))->?R]@\btp.

?X = 1
?R = [a, c, b(2)]

?X = 2
?R = [a, b(1), c]
```

- `\list[|delete(\object) => \list|]`
  Delete all occurrences of the object in list. Selection is made using `==`, not unification. For instance,

```
?- [a,b(1),c,b(?X),b(1)][delete(b(?X))->?R]@\btp.

?X = ?_h4800
?R = [a, b(1), c, b(1)]
```

  Note that `b(?X)` was deleted, but not `b(1)`. On the other hand,

```
?- "[a,b(1),c,b(1)]"^^\list[delete(b(1))->?R]@\btp.

?R = [a, c]
```

  i.e., all occurrences of `b(1)` are deleted.

- `\list[|append(\list) => \list|]`
  Appends one list to another and returns the resulting list.

- `\list[append(\list) => \list]`
  Here the signature is attached directly to `\list` as an object. This means that the `append` method applies directly to class `\list`. In this case, it takes a *list of lists* and returns the list that is the result of appending the lists found in that argument list of lists. For instance

  ```
  ?- \list[append(["[a,b]"^^\list,[c,d],"[e,f]"^^\list])->?R]@\btp.

  ?R = [a, b, c, d, e, f]
  ```

- `\list[|ith(\integer) => \object|]`
  Given a list, returns the object in the ith position. If the position is a variable, returns the position in the list at which the result-object is found. If both the position and the object are variables, enumerates all elements in the list and their position number.

- `\list[|length => \long|]`
  Computes the length of the list.

- `\list[|reverse => \list|]`

- `\list[|toTuple => \object|]`
  Converts lists to tuples. For instance, `[a,b,c][toTuple->(a,b,c)]@\btp`. The primary goal of such a conversion is typically to convert a list of logical statements into a conjunction of these statements, which can then be evaluated as a query. A consequence of this intent is that $\mathcal{F}$LORA-*2*'s singleton tuple like `(foo)` is the same as `foo` and there is no empty tuple, so the query `[][toTuple->?Result]@\btp` returns no results.

- `\list[|sort => \list|]`

- `\list[|=> startsWith(\list)|]`

- `\list[|=> endsWith(\list)|]`

- `\list[|=> subset(\list)|]`
  True if the list object contains the argument list without regard to the order (i.e., treating lists as sets). This method has a convenient shortcut in $\mathcal{F}$LORA-*2*, `\subset`. For instance,

  ```
  ?- [3,2,1] \subset [1,4,5,3,2,9].  // true
  ```

Other methods:

- `\list[|toSymbol => \symbol|]`

- `\list[|=> equals(\object)|]`

- `\list[|typeName => \symbol|]`

Examples:

- `[a,b,c]`
- `[a,b|?X]`
- `[a,b,c|[d,e]]`
- `"[a,b,c]"^^\list`
- `"[a,b|?X]"^^\list`
- `"[a,b,c|[d,e]]"^^\list`

As in Prolog, the part of a list term that follows the bar `|` represents the tail of the list.

39.14 Character Lists

*F*LORA-*2* character lists, *charlists*, are represented as `"..."^^\charlist`. Since character lists are ... lists, they can also be represented using the list notation. For instance, `[102,111,111]` is the same as `"foo"^^\charlist`. The main reason for the existence of the `\charlist` data type is that writing `"foo"^^\charlist` is a lot easier than consulting the ASCII table to find the numeric code for each character in order to write `[102,111,111]`. In addition, *expert* users can simply write `"foo"`, but this syntax is disabled by default because novice users tend to not understand this data structure and misuse it in various ways.

Do not confuse character lists with symbols: symbols are *not* lists and have a completely different representation and one should never use charlists in place of symbols. Character lists are useful in situations when it is necessary to parse the contents of a sequence of characters.

Escape sequences and Unicode are recognized inside *F*LORA-*2* charlists similarly to *F*LORA-*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.

Instance methods. All methods applicable to the `\list` datatype are also applicable to charlists. In addition, some methods applicable to the datatype `\string` also apply to character lists:

- `\charlist[|substring(\integer,\integer) => \charlist|]`
- `\charlist[|toUpper => \charlist|]`

- `\charlist[|toLower => \charlist|]`
- `\charlist[|concat(\charlist) => \charlist|]`

Class methods. The usual class-level methods likewise apply to charlists:

- `\charlist[=> isTypeOf(\object)]`
- `\charlist[toType(\charlist) => \charlist]`

39.15 Special Classes for Callable Literals

In addition to the above, *F*LORA-*2* provides the following builtin meta-classes:

- `\modular` — this is a class for atomic formulas whose truth value depends on the module. This includes F-logic molecules, HiLog predicates, and Prolog predicates declared as `:- prolog` or `:- table`. For instance, `${a[b->c]}:\modular` is true.
- `\callable` — this class includes all atomic formulas that can possibly have a truth value in *F*LORA-*2*. It includes `\modular` formulas as well as various `@\prolog` formulas and builtin primitives like `isinteger{...}`.
  `\callable` does exclude terms that, by their semantics, are not supposed to be truth-valued. These include HiLog terms (as opposed to HiLog predicates), datatype constants, and builtin class names. For instance, `${p(b,?X)}:\callable` and `isinteger{...}:\callable` are true, but `p(b,?X):\callable` and `"abc"^^foobar:\callable` are false. The former is false because `p(b,?X)` in a HiLog term, not a predicate, and the latter is false because `"abc"^^foobar` is a datatype constant.

In both cases, if the class member being tested is a variable then the result is `\undefined` unless this is a typed variable of the matching type. For instance, `?X:\modular` is `\undefined`, but `?X^^\callable:\callable` is `\true`.

39.16 User-defined Types

*F*LORA-*2* also supports *user-defined types*. A user-defined type can be any atom, say `foo`, that is not reserved for the builtin types, i.e., is not prefixed with a "\". A literal of a user-defined type, such as `foo`, has the form `"`*some string*`"^^foo`, i.e., it has the same form as the built-in data types. However, *F*LORA-*2* does not prescribe the contents of the data type and, at present, there is no hook to let the user plug in a personal parser to sort out which literals belongs to the data type and which do not.

Typed variables, introduced in Section 11, also work with user-defined types. Since `foo` may denote a class, a user-defined type, or both, such a variable, `?X^^foo` binds to *both* the members of the class `foo` and the literals of type `foo`. For example,

```
{a,b}:foo.

?- a=?X^^foo, b=?Y^^foo.      // true
?- "abc"^^foo=?X^^foo.        // true
?- d=?X^^foo.                 // false: d is not in class foo
?- "abc"^^moo=?X^^foo.        // false: type mismatch
?- "abc"^^foo=?X^^(foo;moo).  // false: don't use foo both as a class & type
?- insert{"abc"^^foo:foo}, "abc"^^foo=?X^^(foo;moo).  // true
```

However, it is not recommended to use the same symbol both as a class and as a user-defined type at the same time in the same module. For example, in the next-to-last example above, `foo` is used as a class in the expression `?X^^(foo;moo)`, so it will not unify with the literal of type `foo` unless `"abc"^^foo` is a member of the class `foo`. That last possibility is illustrated by the last line in the above example.

40 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 method `Meth`. The second expression is a class-level statement, so it applies to all members of `C1` (now viewed as a class) and to all subclasses of `C1`.

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

For example,

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

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 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`.

41 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), \false.
```

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 does not explicitly catch it, the exception will eventually be caught by the $\mathcal{F}$LORA-*2* command loop. In the above example, 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 frames, not just predicates. However, `?Error` — both in `catch` and in `throw` — must be HiLog or Prolog terms. No frame literals 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 frame is reified.

Some exceptions are thrown by $\mathcal{F}$LORA-*2* itself, and applications might want to catch them:

- `'_$flora_undefined'(?MethodSpec,?ErrMsg)` — thrown when undefinedness checking is in effect (see Section 43.1) and an attempt is made to execute an undefined method or predicate. The first argument in the thrown exception is a specification of the undefined predicate or the method that caused the exception. The second argument is the error message.
- `'_$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 rule 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 46.3.

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 that the applicable files include `flora_exceptions.flh` 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 of the exceptions thrown by Prolog (which can be found in the manual). These exceptions have the form

`error(`*errortype*`(`*arguments*`),context(`*Message*, *Backtrace*`))`

However, FLORA-2 aims to intercept all Prolog exceptions and contextualize them in the appropriate FLORA-2 terms and any non-caught Prolog exception should be treated as an omission to be fixed in the next release.

42 The Compile-time Preprocessor

FLORA-2 supports a C-style preprocessor, which is invoked during the compilation. The most important commands are

```
#define variable value
#define macro(arg_1,...,arg_n) expression
#ifdef variable
#ifndef variable
#else
#endif
#include "file"
```

There are many features that go beyond the C preprocessor such as the tests

```
#if exists("file")
#if !exists("file")
```

and many others. For example, it is possible to enable macro substitution inside quotes. An advanced user is referred to the XSB manual where this preprocessor, called gpp, is described in an appendix to Manual 1.

43 Debugging User Knowledge bases

FLORA-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 8.1). When a problem is deemed serious enough, errors are reported.

The **most important rule** in debugging FLORA-2 knowledge bases is: **never ignore any kind of warnings** issued by the system. This golden rule actually applies to programming in *any* language. In addition, it is possible to tell FLORA-2 to perform various run-time checks, as described below.

43.1 Checking for Undefined Methods and Predicates

$\mathcal{F}$LORA-*2* has support for checking for 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 user 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 that this 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 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 $\mathcal{F}$LORA-*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 *existing* 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.

Note: `Method[mustDefine(on)]@\sys` does not apply to *newly-created* modules and `Method[mustDefine(on(foomodule))]@\sys` will result in an error if `foomodule` does not exist.

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 $\mathcal{F}$LORA-*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, $\mathcal{F}$LORA-*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.
```

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 off in *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 parentheses 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{...}` built-in. For instance, suppose `FOO` is a predicate or a frame 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 frame 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 checks and meta-programming. We should note one subtle interaction between these checks and meta-programming. Suppose the user knowledge base does not have any class membership facts and the undefinedness checks are turned on. Then the meta-query

```
?- a:?X.
```

would cause the following error:

++Error[Flora-2]: Undefined class ?:? in user module main

Likewise, if the knowledge base does not have any method definitions, the query `?- ?X[?Y->?Z].` would cause an error. This might not be what one expects because the application 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 "weird" facts into the knowledge base and special-case them. For instance, one could add the following facts to silence the above errors:

```
ads_asd_fsffdfd : ads_asd_fsffdfd.
ads_asd_fsffdfd[ads_asd_fsffdfd -> ads_asd_fsffdfd].
```

The user can then arrange the things so that anything that contains `ads_asd_fsffdfd` would be discarded.

Another way to circumvent the problem is to turn the undefinedness checks off temporarily. For instance, suppose the query `?- ?X:a` causes an unintended undefinedness error in module `foo`. Then we can avoid the problem by posing the following query instead:

```
?- 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{...}` built-in (note the curly braces):

```
#include "flora_exception.flh"

?- catch{?X:a, FLORA_UNDEFINED_EXCEPTION(?,?)@\prolog, true}.
```

Undefinedness checks 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 and 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 `?- 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.

43.2 Type Checking

Although $\mathcal{F}$LORA-*2* allows specification of object types through signatures, type correctness is not checked automatically. A future version 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 knowledge engineer has to do some work to make type checking happen. For instance, one 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], \naf ?R:?D
       ;
       %% Defined methods that do not have type information
       ?O[?M->?R], \naf ?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 knowledge base, 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 knowledge base 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, \naf ?R:?D@?Mod1
       ;
       %% Defined methods that do not have type information
       (?O[?M->?R], \naf ?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` (which can be abbreviated to `\tpck`). Its syntax is:

```
?- Type[check(?Specification,?Result)]@\typecheck.
?- Type[check(?Specification,?Result)]@\tpck.
```

The `?Specification` variable must be bound to a base frame, as described below. `?Result` gets bound to the evidence of type violation (one or two atoms that violate the typing constraint).

- If `?Specification` 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 an unbound variable, then the constraints are checked in all modules. `?Meth` can also be a variable. In this case all non-transactional methods will be checked.
- If `?Specification` 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 unbound variables.

- If `?Specification` is of the form `?[|?Meth->?|]@?Mod` then only the consistency between `->` and `=>` is checked and only for frames that are statements about classes as a whole, i.e., the frame formulas of the form `?[|?Meth->?|]@?Mod` and `?[|?Meth=>?|]@?Mod`. The `obj[...]`-style frames are ignored. Missing types (semistructured data) are flagged.
- If `?Specification` is of the form `?[|?Meth=>?|]@?Mod` then again only the consistency between `?[|?Meth->?|]@?Mod` and `?[|?Meth=>?|]@?Mod` is checked, but missing types are not flagged.

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]}), ${a[foo -> e]}]
```

The first item in the list (the pair inside parentheses) means that the frame `a[b -> e]` violates the type constraint specified by the signature `a[b => d]`. The second item means that the frame `a[foo -> e]` 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 constraint 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 constraint represents what kind of type checking is to think that `=>` represents typing and, therefore the `=>`-style constraints mean that only the methods that have typing information will be type-checked. The `->`-style constraints, on the other hand, mean that all methods will be checked—whether they have signatures or not. Similarly, `...[|...|]`-style constraints indicate that only information about classes as a whole will be type-checked, while information specified explicitly for individual objects will not be. In contrast, `...[...]`-style constraints indicate that all type information will be verified.

43.3 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 terminating query that will flag a violation of a cardinality constraint if and only if one exists.

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 built-ins in their bodies. Therefore, in practice, the user should do 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, in *F*LORA-*2* it is easy to verify that a particular method 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:

```
?- 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 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 least 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 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 an atomic signature frame (no need to specify reification ${...}). The method type of the signature can be only `=>`, but the frames can have the `...[...]`-style or `...[|...|]`-style. The former checks the cardinality constraints of object methods, while the latter checks cardinality constraints only for default values of the methods.

- `Cardinality[check(?Object[?Method => ?])]@\typecheck`
  Checks cardinality constraints for `?Method` of type `=>` in the current module. That is, whether there are instances of the literal `?Object[?Method -> ?Val]` that violate a cardinality constraint imposed by some signature of the form `?Object[?Method{?Low..?High}=>?Type]` (which may be a derived signature).

- `Cardinality[check(?Obj[|?Method =>?|]@?Module)]@\typecheck`
  Checks cardinality constraints for the default values of `?Method` in module `?Module`. If `?Module` is unbound and a cardinality constraint violation is detected in some module then `?Module` is bound to that module. That is, it is a check for whether there are instances of the literal `?Object[|?Method -> ?Val|]` that violate a cardinality constraint imposed by some signature of the form `?Object[|?Method{?Low..?High}=>?Type|]` (which may be a derived signature).

- `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 knowledge base, 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 `cl`) 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
  ```

43.4 Logical Assertions that Depend on Transactional and Non-logical Features

On page 148 we mentioned the potential problems when tabled predicates or frames 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 knowledge base:

```
?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, transactional and side-effectfull 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-effectfull statements, but it does not warn about all Prolog predicates of this kind. Therefore, caution needs to 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.

43.5 Examining Tables

Sometimes it is useful to be able to examine the tables that XSB has generated while answering queries. To this end, $\mathcal{F}$LORA-*2* provides the following library predicates:

```
\tabledump(File,AtomicGoal)
\tabledump(File,AtomicGoal,Option)
```

The `File` argument specifies the file in which to place the results. The results are in the $\mathcal{F}$LORA-*2* format, as explained below. If `File` is `userout` then the results go to the standard output. `AtomicGoal` is a HiLog predicate or an atomic frame (e.g., `?[?->foo]`). It specifies the subgoals for which tables are being requested. `Option` is the option selected. Currently three options are supported: `summary` (minimalist output that summarizes the overall statistics of tables), `subgoals` (more details about individual subgoals). The third option, `answers`, will output full details, including the information about each called subgoal and all answers to all subgoals.

The first (binary) form of `\tabledump` above is equivalent to `\tabledump(File,AtomicGoal,summary)`.

In all three cases, `\tabledump` generates information about the tables for subgoals that are subsumed by `AtomicGoal`. If `AtomicGoal` is a variable, information is displayed about all tables.

When the `summary` option is used, the information is displayed in the following format:

```
AtomicGoal[total_subgoals->..., total_subgoal_answers->...].
```

For instance,

```
?- \tabledump(userout,?).

${?A(?B)}[total_subgoals->1, total_subgoal_answers->0].
${?A(?B,?C)}[total_subgoals->3, total_subgoal_answers->2].

?- \tabledump(userout,p(2,?)).

${p(2,?A)}[total_subgoals->1, total_subgoal_answers->1].
```

If the third option is used (`answers`), then in addition to the output produced for the `summary` option the system will show the individual subgoals subsumed by `AtomicGoal` and the answers to each:

```
AtomicGoal[
   total_subgoals->1,
   subgoal_details->{Subgoal1[total_answers->..., answer_list->[answer1,...],
                     Subgoal2[total_answers->..., answer_list->[answer2,...],
                     ...}].
```

For instance, the following might be produced for the two earlier requests, if the `details` option is specified :

```
?- \tabledump(userout,?,answers).

${?A(?B)}[total_subgoals->1, total_subgoal_answers->0].
${?A(?B)}[total_subgoals->1,
          subgoal_details->{${q(?_h34)}[total_answers->0, answer_list->[]]}].
${?A(?B,?C)}[total_subgoals->3, total_subgoal_answers->2].
${?A(?B,?C)}[total_subgoals->3,
```

```
            subgoal_details->{${p(2,b)}[total_answers->1, answer_list->[${p(2,b)}]],
                              ${p(1,a)}[total_answers->1, answer_list->[${p(1,a)}]],
                              ${p(?_h65,?_h67)}[total_answers->0, answer_list->[]]}].

   ?- \tabledump(userout,p(2,?),answers).

${p(2,?A)}[total_subgoals->1, total_subgoal_answers->1].
${p(2,?A)}[total_subgoals->1,
           subgoal_details->{${p(2,b)}[total_answers->1, answer_list->[${p(2,b)}]]}].
```

If the second option is used (*subgoals*), then the output is similar to the third option, but the `answer_list` part is not shown. In this case, the output is slightly smaller. However, it should be kept in mind that for large knowledge bases with large numbers of answers table dumps can be huge (hundreds of megabytes) and it can take considerable time to dump these tables. In this case only the first option (and maybe the second, if you must) is recommended.

43.6 Examining Incomplete Tables

Sometimes it becomes necessary to examine incomplete tables (i.e., tables to subgoals that have not yet been completely evaluated) in the middle of execution or upon exception.

To get a dump of all incomplete tables in the middle of the computation, one has to insert the predicate `\dump_incomplete_tables` in an appropriate place in user's rule bodies. For instance,

```
r(0,?):- !, \dump_incomplete_tables(temp).
r(3,?A):- r(5,?A).
r(?N,?A):- ?N1 is ?N - 1, r(?N1,?A).
?- r(5,foo(a)).
```

will put the following into the file `temp`:

```
${r(5,foo(a))}[scc->1].
${r(4,foo(a))}[scc->1].
${r(3,foo(a))}[scc->1].
${r(2,foo(a))}[scc->2].
${r(1,foo(a))}[scc->3].
${r(0,foo(a))}[scc->4].
```

It says that there are four strongly connected components of subgoals, numbered 1, 2, 3, and 4. All of these subgoals are still waiting to be fully computed, but at the moment their truth values are still unknown.

More often, though, one might need to examine incomplete tables after an exception, if the user suspects that the exception has something to do with tabled subgoals. Such a table dump looks exactly like the dump produced by `\dump_incomplete_tables/1` but it is requested differently. First, one must execute the query

```
?- \set_dump_incomplete_tables_on_exception.
```

Then, after an exception took place, the user should execute the query

```
?- \dump_incomplete_tables_after_exception(filename).
```

For instance,

```
?- \set_dump_incomplete_tables_on_exception.
q(0,?):- !, abort@\sys.
q(3,?A):- q(5,?A).
q(?N,?A):- ?N1 is ?N - 1,q(?N1,?A).
?- q(5,foo(a)).
?- \dump_incomplete_tables_after_exception(temp).
```

The dump of the tables that were incomplete at the time of the abort will be in the file `temp` and will have the same structure as before:

```
${q(5,foo(a))}[scc->1].
${q(4,foo(a))}[scc->1].
${q(3,foo(a))}[scc->1].
${q(2,foo(a))}[scc->2].
${q(1,foo(a))}[scc->3].
${q(0,foo(a))}[scc->4].
```

$\mathcal{ERGO}$, in addition, has a much more useful primitive, `showgoals{...}`, which can show the information about incomplete computations without the need to modify the program rules and in a more focused way.

43.7 Tracing Tabled Calls via Forest Logging

While the regular Call-Redo-Exit-Fail logging is useful in many cases, it is extremely slow and generates large amounts of output. A query that runs mere 10 seconds can take hours to execute

under tracing and it may generate hundreds of megabytes worth of trace output. Clearly, it is hard to use the regular tracing facility under such conditions. On top of this, even though the regular trace is capable of producing queriable output, such trace does not supply parent-child relationships between calls, and, due to this, automatic analysis of such trace is very hard.

Fortunately, there is an alternative: *forest logging*. Forest logging is a kind of tracing that keeps track only of tabled predicate calls. It is very fast (time overhead is less than 80% compared to orders of magnitude for the regular trace), it generates drastically smaller trace logs, and it preserves the parent-child relationship between the calls. The drawback is that logforest traces track tabled calls only, but in $\mathcal{F}$LORA-*2* this is not a serious problem since most calls that are of interest to the user are tabled (except for transactional predicates and methods).

$\mathcal{F}$LORA-*2*'s forest logging is implemented as a presentation layer on top of XSB's forest logging. Its format and other details are described next.

Forest logging. To start forest logging, the user must issue the command `\logforest` at the $\mathcal{F}$LORA-*2* prompt or include the query `?- \logforest` where appropriate. In the latter case, logging will start after this subgoal gets executed. To stop forest logging, issue the command `\nologforest`.

The entries in the log represent the following actions that occur during tabled evaluation of queries and every entry has an Id which is a non-negative integer.

- *A call to a tabled subgoal.* When a call to a tabled subgoal $\mathtt{S}_1$ is made from a derivation tree for $\mathtt{S}_2$, a frame literal is recorded in the following format:

  ```
  call(Id)[goal->S1, stage->Stage, parent->S2].
  ```

  where `Id` is the generated Id of the call and `Stage` is

  - `new` if $\mathtt{S}_1$ is a new subgoal.
  - `comp` if $\mathtt{S}_1$ is an old *completed* subgoal.
  - `incmp` if $\mathtt{S}_1$ is an old *incomplete* subgoal (i.e., it has not been fully evaluated yet).

  If $\mathtt{S}_1$ is the first tabled subgoal in an evaluation, $\mathtt{S}_2$ is represented by the atom `null`. If the call is negative, a similar fact of the form `negative_call(Id)[goal->S1, stage->Stage, parent->S2]` is logged.

- *Derivation of a new answer.* When a new unconditional answer `A` is derived for subgoal `S` and added to the table, the following fact is logged:

  ```
  answer(Id)[goal->S, answer->A].
  ```

As before, `Id` is the identity number generated for this particular action.

When a new conditional answer `A` is derived for subgoal `S` and the delayed literals are `D`, a log of the form `conditional_answer(Id)[goal->S, answer->A, delayed_literals->D]` is recorded.

- *Return of an answer to a consuming subgoal.* When an answer A is computed and returned to a consumer subgoal `S` in a derivation tree for `ST` and the table for `S` is incomplete, the following fact is recorded:

  ```
  answer_to_consumer(Id)[goal->S, answer->A, consumer->ST].
  ```

  If `A` is conditional, this entry `delayed_answer_to_consumer(Id)[goal->S, answer->A, consumer->ST]` is recorded.

- *Delaying a negative literal.* When a selected negative literal `N` of a node `S` is delayed due to its involvement in a loop through negation, and `S` is in a derivation tree for $\mathtt{S}_\mathtt{T}$, a fact of this form is logged.

  ```
  delay(Id)[delayed_literal->N, parent->ST].
  ```

- *Subgoal completion.* When a set $\mathcal{S}$ of subgoals is completely evaluated, for each $\mathtt{S} \in \mathcal{S}$ a fact of the following format is logged for each `S`:

  ```
  completed(Id)[goal->S, sccnum->SCCNum].
  ```

  Here `SCCNum` is the identifier generated for the set of subgoals $\mathcal{S}$. If `S` is completed early, `SCCNum` is the atom `ec`.

- *Table abolishes.* There are three occasions where tables are abolished.

  – When a tabled subgoal `S` is abolished, a fact of the following form is logged:

    ```
    table_abolished(Id)[type->subg, goal->S].
    ```

  – When all tables for a predicate p/n are abolished, a fact of the following form is logged:

    ```
    table_abolished(Id)[type->pred, goal->[p/n]].
    ```

  – When all tables are abolished, the following fact is logged:

    ```
    table_abolished(Id)[type->all].
    ```

- *Recording of errors.* If an error is thrown and the execution is in a derivation tree for subgoal `S`, forest logging records the following fact:

```
error(Id)[goal->S].
```

By default, logs are sent to the current output stream. However, it is usually more convenient to dump the logs to a file using the following command: `\logforest(File)`. For instance, executing

```
?- \logforest('foobar.flr').
```

will direct *FLORA-2* to dump the entire forest log into the file `foobar.flr`. In case the user also wants to examine the original XSB's forest logging trace, the following command can be executed: `\logforest(FloraTraceFile,XSBTraceFile)`. For instance, executing

```
?- \logforest('foobar.flr', 'foobar.P').
```

will dump the trace in the above *FLORA-2* format into the file `foobar.flr` and keep the original XSB's forest log in the file `foobar.P`.

We may want to skip certain types of log entries in some circumstances such as reducing log file sizes. *FLORA-2* provides `\logforest(HideOptions)` and `\logforest(File, HideOptions)`, where `HideOptions` is a list of log types to be skipped. The elements of `HideOptions` can be one of the following: `call, negative_call, delayed_call, answer, conditional_answer, answer_to_consumer, delayed_answer_to_consumer, completed, table_abolished`, and `error`. For instance,

```
?- \logforest([table_abolished, error]).
?- \logforest('foobar.flr', [table_abolished, error]).
```

will not record logs for table abolish and error actions.

Note that `\logforest(FloraTraceFile,XSBTraceFile)` and `\logforest(File,HideOptions)` will not be mixed since *FLORA-2* checks whether their second argument is a list or not to tell which command is executed. Similarly, `\logforest(HideOptions)` and `\logforest(File)` will not be mixed.

It is important to keep in mind, however, that *no* output is produced (to the file or output stream) unless the user issues the command `\nologforest`. In other words, forest logging can be obtained only *after* the evaluation is finished. If you expect the query to throw an error, it is a good idea to use the *FLORA-2* `catch...` primitive. If the query does not terminate, you should also wrap the query with the `timed_call/3` XSB predicate.[23]

[23] `timed_call/3` is described in the XSB Manual, Part 1. Do not forget to reify the query argument to `timed_call/3`.

Low-level forest logging. Sometimes it is necessary to look into the guts of forest logging without converting it into the FLORA-2 format. Typically this is needed for low-level debugging of FLORA-2 itself. The command for this is `\logforestlow`; it directs FLORA-2 to display forest logs directly in the XSB format. There is also a version of this command for saving the log in a file:

```
?- \logforestlow('foobar.flr').
```

As before, one should remember to issue the command `\nologforest` in order to flush the log to the output. FLORA-2 also provides `\logforestlow(HideOptions)` and `\logforestlow(File, HideOptions)` to skip certain types of low-level log entries.

43.8 Controlling Subgoal and Answer Size, Timeouts, Unification Mode

Sometimes it is useful to be able to control the term-size of the subgoals that can be generated during evaluation and the size of the answers returned. The former is useful if the user knowledge base has recursive rules in which the size of the body literals is greater than the size of the head. The latter is useful when a query has an infinite number of answers. In both cases, limiting the size can terminate a run-away computation. Timeouts are useful when it is desirable to stop the computation if it does not finish within a preset amount of time. To control these features, FLORA-2 provides the builtin primitive that can appear in any query or rule body:

```
?- setruntime{Opt1,Opt2,...}
```

Several options can be used in the same `setruntime` command. These options are described below.

43.8.1 Timeouts

The following `setruntime{...}` options can be used to control various types of timeouts:

```
timeout(Spec)
timeout(Spec,Spec)
timeout(0)
```

where *Spec* is either `max(Time,Handler)` or `repeating(Time,Handler)`. Here *Time* is a positive integer that specifies the number of seconds after which queries should be interrupted. The first form will interrupt the queries once (so this form is used to specify timeouts) and the second will interrupt queries periodically, after each *Time* seconds. Either `max`, or `repeating`, or both can be specified (via the 1-argument and 2-argument form of `timeout`). In either case, *Handler* will

be called at each interrupt. The last form above removes all timeout restrictions. Note that each subsequent `setruntime` timeout-setting command will override the previous one.

Here are some examples:

```
?- setruntime{timeout(max(4,fail))}.
?- setruntime{timeout(max(100,abort))}.
?- setruntime{timeout(repeating(4,Handler(?)),max(100,abort))}.
?- setruntime{timeout(0)}.
```

Note that once any of these commands is issued, it applies to all subsequent queries. The last command resets the timeout to infinity.

Handler in the `max` and `repeating` specifications must have one of the following forms:

- A predefined error handler: `ignore` (ignore the timer interrupt), `abort` (abort the current goal), `fail` (make the current goal fail); or
- A Prolog predicate defined in some Prolog module; or
- A predicate declared using $\mathcal{F}$LORA-*2*'s `:- prolog{...}` directive.
- A predefined error handler `pause`, which is available only **in** $\mathcal{ERGO}$. In that case, the execution pauses and the user is given the opportunity to inspect the state of the system and then either continue or abort the computation.[24] The `pause` interrupt handler is perhaps the most useful when the system runs in interactive mode.

If *Handler* is a Prolog predicate that exists in the default Prolog's `usermod` module then the handler should look like `foobar(?)@\prolog` or `foobar@\prolog` (i.e., one or zero arguments). The argument, if present, must be an unbound variable, which will be bound at runtime to the goal being interrupted. A Prolog handler is specified as `foobar(?)@\prolog(`*`mod`*`)` if `foobar/1` is in Prolog's module *mod* rather than in the `usermod` module. In both of these cases, `foobar/1` must be defined in a Prolog program that $\mathcal{F}$LORA-*2* has already loaded. In $\mathcal{ERGO}$, information primitives such as `showgoals{}`, can also be used as handlers.

If the handler is an $\mathcal{F}$LORA-*2* predicate declared as `:- prolog{...}` then it must be specified as `foobar(?)@`*`mod`* (note: not `\prolog(`*`mod`*`)`, but *mod*), where *mod* is an $\mathcal{F}$LORA-*2* module (or `\@`, if the current $\mathcal{F}$LORA-*2* module is desired)—see Section 17.3.

For instance, the following fragment in $\mathcal{F}$LORA-*2* sets the system for periodic timer interrupts:

[24] The ability to pause a computation exists only in $\mathcal{ERGO}$ and this is why the pause interrupt handler is available only there.

```
:- prolog{periodic_handler/1}.
periodic_handler(?Goal) :- writeln(interrupted_goal=?Goal)@\prolog.
?- setruntime{timeout(repeating(2,periodic_handler(?)@\@))}.
```

In this example, the FLORA-*2* predicate `periodic_handler/1` will be called every two seconds.

Alternatively, one could define `periodic_handler/1` as a Prolog predicate in a `.P` file, e.g., `foo.P`, and then set periodic interrupts as follows:

```
?- ['foo.P'].
?- setruntime{timeout(repeating(2,periodic_handler(?)@\prolog))}.
```

Care must be taken to not obliterate periodic handlers during loading. For instance, in both of the above cases `periodic_handler/1` may get erased. In case of a Prolog program (the second example), the danger is not as great: `periodic_handler/1` may get replaced only if another Prolog file gets loaded and happens to have the same predicate. A Prolog predicate may also be explicitly abolished by the user, but this is rare. In case of periodic handlers defined in FLORA-*2* modules, however, the problem occur more frequently. For instance, the first example may be loaded into some module, `foo`, and then, due to an oversight, something else may get loaded (as opposed to "added") into the same module. In that case, `periodic_handler/1` will be obliterated and the periodic timer interrupt specification will become orphaned, resulting in a runtime error. If this happens, errors will be issued whenever the interrupt handler is called.

43.8.2 Subgoal Size Control

The following options provide subgoal size control:

```
goalsize(TermSize,abort)
goalsize(TermSize,abstract)
goalsize(TermSize,pause)          // Ergo only
```

These options control the maximum size of the subgoals called during the evaluation. *TermSize* specifies the max size of the terms returned as answers. This takes into account both nesting of all function symbols except lists (lists are considered to be of size 1) and the arity (width) of the terms.

The first form of the above options will abort the query if the specified limit is reached. The second form will *perform call abstraction* and replace the deeply nested subterms with new variables. The `pause`-action will pause the computation and allow the user to examine the state of the computation before deciding what to do next. Examples:

```
?- setruntime{goalsize(100,abort)}.
?- setruntime{goalsize(100,abstract)}.
```

The above options are incompatible, so each subsequent option overrides the previous one.

43.8.3 Answer Size Control

The following options enable answer size control:

```
answersize(TermSize,abort)
answersize(TermSize,abstract)
answersize(TermSize,pause)          // Ergo only
```

The meaning of the `abort` options is the same as for the goal size. The meaning of the `abstract` options is that the size of terms is limited to the specified numbers. If higher size answer-objects are generated, their truth value is set to *undefined*. This is called *answer abstraction*. For instance, if the computation generates the answers `p(f(a))`, `p(f(f(a)))`, etc., then answer abstraction at size 5 will generate the answers up to `p(f(f(f(a))))` and the last answer, `p(f(f(f(f(?)))))`, will be *undefined*. This means that some of the instances of that last answer might be true and some false.

Examples:

```
?- setruntime{answersize(100,abort)}.
?- setruntime{answersize(100,abstract)}.
```

43.8.4 Controlling the Number of Active Goals

The following options enable one to control the number of active (i.e., not completely evaluated) tabled subgoals during query evaluation, which is useful for termination and performance analysis.

```
activegoals(TermSize,abort)
activegoals(TermSize,pause)          // Ergo only
```

The meaning of the `abort` and `pause` actions is the same as in the case of subgoal and answer size controls.

43.8.5 Memory Usage Limit

The following option is available:

```
memory(memory-limit-in-GBs)
```

For instance,

```
?- setruntime{memory(12)}.
```

will set the limit to 12 GBs. If the $\mathcal{F}$LORA-*2* process exceeds this amount, the computation will be aborted.

$\mathcal{ERGO}$ (but not $\mathcal{F}$LORA-*2*) also supports the following:

```
memory(memory-limit-in-GBs,Action)
```

where *Action* is either `abort` or `pause`. The `abort` option works exactly like `memory(Size)`, while if `pause` is given then the computation will pause instead of aborting. The user can then ask informational queries about the system state and decide what to do. For instance, one may decide whether to abort or to increase the memory limit and continue.

43.8.6 Unification Mode

The default unification mode in $\mathcal{F}$LORA-*2* is unsound, in general. That is, non-unifiable terms might be unified, as explained below. The following runtime options can be used to change the unification mode:

```
unification(fast)
unification(pedantic)
```

For instance,

```
?- setruntime{unification(fast)}.
?- setruntime{unification(pedantic)}.
```

These commands controls the experimental feature of unification mode. The default unification mode is `fast`. This is a logically unsound mode: it does not do the occurs-check and so the unification `?X = f(?X)` will succeed even though the two terms are not unifiable in the classical sense. Under the `pedantic` mode, such a unification would fail.

Unifying in the fast mode can lead to hard-to-find errors, although FLORA-2 makes attempts to limit the damage and the probability of serious problems occurring due to this mode of unification is low.

43.8.7 Suppressing the Unsafe Negation Warning

Evaluation of some queries may encounter situations where FLORA-2 is unable to determine the real truth value of an answer and will report it as undefined. When this happens, a warning such as this will be issued:

```
++Warning[FLORA-2 ]> File: ..., line: ...
 Subgoal: (\naf (q(?A,?B), (\naf p(?A))))
 Unbound variables: [?B]
 The subgoal has unbound variables under \naf or 'forall'.
 Ergo is unlikely to evaluate this to true or false.
 Try to bind free & quantified variables to finite domains.
 If everything is correct, this warning can be suppressed with
    setruntime{unsafe_naf(ignore)} or
    setruntime{unsafe_naf(ignoreonce)}.
```

This occurs when an unbound variable is found during the evaluation of `\naf` or `forall` (which translates into `\naf exist(...)^(\naf ...)` and thus also involves negation), and is called *unsafe negation \(naf)*. In the above case, the variable `?B` in `q/2` will be unbound during the evaluation of the outer `\naf`, whence the warning.

As suggested in the warning, it can be suppressed either globally, by issuing the command

```
?- setruntime{unsafe_naf(ignore)}.
```

or just for the duration of a single query:

```
?- setruntime{unsafe_naf(ignoreonce)}, the-query-of-interest.
```

Of course, these warnings should not be suppressed willy-nilly but only when the user understands that `the-query-of-interest` involves unsafe negation and there is no better formulation for that query.

43.8.8 Strict `setof` Mode

The strict `setof` mode affects the way the `setof` aggregate function operates. As explained in Section 30, a query like `?Result = setof{?X|condition(?X)}` obtains all the `?X` that satisfy `condition` and then sorts the results to eliminate duplicates. This is not a complete description, however. For example, the sort function considers `f(?I)` and `f(?J)` (i.e., terms that are isomorphic up to a variable renaming) as being different. In addition, identical reified terms like `${d(c)}` and `${d(c)}` may also be considered different because it takes an extra step to tell them apart. Since reified and non-ground terms occur in aggregate primitives rather infrequently, expanding the overhead to eliminate duplicates among such terms is unjustified. If it is nevertheless desired to do such elimination, the user can request such a service by executing the query

```
?- setruntime{setof(strict)}.
```

This can be also executed as a subgoal in a bigger query or a rule body. To return back to the regular mode, execute

```
?- setruntime{setof(lax)}.
```

43.9 Non-termination Analysis

There are two main non-termination causes in $\mathcal{F}$LORA-*2*: infinite number of calls and infinite number of answers, and both of them arise due to recursive rules together with function symbols. Since non-termination is normally very hard to debug, $\mathcal{F}$LORA-*2* provides a tool, called a non-*Termin*ation Anal*yzer*, or the `Terminyzer`, to help the user to locate and understand non-terminating behavior.

When a runaway computation is suspected, the user should stop the execution manually or he could set a time limit or a term-size limit prior to starting the query, as explained in Section 43.8. `Terminyzer` then analyzes the execution's forest logging trace and reports the sequences of tabled unfinished calls and their respective rule ids (the rules in the knowledge base from which these calls were issued) that actually caused the non-termination. Below, we describe the usage of `Terminyzer`; more information can be found in [11, 12].

To start `Terminyzer`, the user issues the following command

```
?- terminyzer(XSBTraceFile,SummaryFile)@\prolog(flrterminyzer).
```

where `XSBTraceFile` is the XSB logforest trace that can be obtained by executing the query `\logforest(`*`FloraTraceFile`*`,`*`XSBTraceFile`*`)`, as described in Section 43.7 and *SummaryFile* is the non-termination analysis summary in the $\mathcal{F}$LORA-*2* loadable format. (Note: the input to

Terminyzer is *XSBTraceFile*, not *FloraTraceFile*.) The entries in the summary file can be used to explain non-terminating behavior, if it has occurred. They are as follows:

- *Logforest file.* The logforest trace file being analyzed is reported as:

  ```
  logfile(XSBTraceFile).
  ```

- *Unfinished tabled calls.* If a tabled subgoal S_1 is called from a derivation tree for subgoal S_2 because of the rule with id `RId` (i.e., S_2 matches this rule's head and S_1 matches one of its body subgoals) and S_1 if is not completely evaluated, the following two frames are reported:

  ```
  unfinished_subgoal(SubgoalId1)[subgoal->S1,ruleid->RId].
  unfinished_call(CallId)[parent->SubgoalId2,child->SubgoalId1,ruleid->RId].
  ```

  where `SubgoalId1` and `SubgoalId2` are unique ids generated for S_1 and S_2. (Note: only one id is generated for each non-variant subgoal.) `CallId` is the unique id generated for this call from S_2 to S_1. Both subgoal ids and call ids are non-negative integers generated in the temporal order of subgoal and call creation during the evaluation.

  Besides, the number of unfinished subgoals and calls are also recorded as follows:

  ```
  number_of_subgoals(unfinished,NumOfSubgoals).
  number_of_calls(unfinished,NumOfCalls).
  ```

- *Non-termination due to infinite number of subgoals.* If the trace is generated by an execution that generates infinitely many subgoals, these subgoals must be generated by a set of recursive predicates that generate subgoals with increasing term-size. Furthermore, each cycle of recursive calls to these predicates produces a set of deeper and deeper subgoals. The id sequence of subgoals produced by the first recursive cycle is recorded in the following form:

  ```
  call_loop[subgoals->SubgoalIdSequence].
  ```

  The number of subgoals in the sequence is reported as:

  ```
  number_of_subgoals(callloop,NumOfSubgoals).
  ```

- *Non-termination due to infinite number of answers.* When the trace is generated by an execution that produces infinitely many answers, these answers are produced by a set of recursive predicates that generate answers of increasingly large term-size. This set of recursive unfinished subgoals and their relationships are reported as follows.

  ```
  answerflow[subgoals->SubgoalIds,calls->CallIds,loop->LoopSubgoalIds].
  ```

where `subgoals` and `calls` are the ids of those unfinished subgoals and calls that are involved in answer generation and `loop` is the sequence of subgoal ids that form the recursive loop. The `loop` also indicates the call-relationship among the subgoals in the sequence. For instance, `[5,6,5]` says that subgoal 5 calls subgoal 6 and subgoal 6 calls subgoal 5, and this recursion is producing infinitely many answers.

The number of recursive subgoals and calls and the size of the loop are recorded as follows:

```
number_of_subgoals(answerflow,NumOfSubgoals).
number_of_calls(answerflow,NumOfCalls).
number_of_subgoals(answerflowloop,LoopSize).
```

The information produced by `Terminyzer` can be used to locate the rules and specific calls that cause non-terminating behavior.

44 Considerations for Improving Performance of Queries

Left-to-right processing. The first rule in improving the performance of $\mathcal{F}$LORA-*2* queries 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, as with databases, Cartesian products should be avoided at all costs.

Nested frames and path expressions. The $\mathcal{F}$LORA-*2* compiler makes decisions about where to place the various parts of complex frames, and the knowledge engineer can affect this placement by writing frames 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 frame should be written in the opposite order. The other consideration has to do with literals that have nested frames in them. For instance, the following query

```
?- ..., ?X[attr1->?Y[attr2->?Z]], f(?P.attr3), ...
```

is translated as

```
?- ..., ?X[attr1->?Y], ?Y[attr2->?Z], f[?P->?_newvar], f(?_newvar), ...
```

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 more efficient code.

Similar considerations apply to `f(?P.attr3)`, but here `?P.attr3` is computed first and the result is passed to the predicate `f/1`. Note that frames and truth-valued path expressions are not allowed as nested arguments in predicates and functions. That is, `f(?Y[attr2->?Z])` or `f(?Y.attr2[])` would cause compiler to issue errors. However, *pure* path expressions, which have no truth values, can be nested inside predicates and functions. For instance, `f(?Y.attr2)` or even `f(?Y[foo->bar].attr2)` are acceptable except inside `insert...` and `delete...` primitives.

As with nested frame literals, 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 databases, statistical information is not available to the $\mathcal{F}$LORA-*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 user may 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:

```
?- ?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:

```
?- ?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.

The delay quantifiers `wish` and `must`, described in Section 35, can also be helpful in ensuring that certain variables are bound.

45 Compiler Directives

45.1 Executable vs. Compile-time Directives

Like a Prolog compiler, the FLORA-*2* compiler can take compiler directives. As 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 the knowledge base. 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 file prior to the first appearance of a rule or a fact, because such directives have effect only after they are found and processed.

To better understand the issue, consider the following simple example (say, in the file `test.flr`):

```
:- op{400,xfx,fff}.
a fff b.
?- ?X fff ?Y.
```

If one loads this example, 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 example were

```
?- 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.

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` (default), `inheritance=monotonic`,
`tabling=reactive` (default), `tabling=passive`,
`tabling=variant` (default), `tabling=subsumptive`,
`custom=none` (default), `custom=`*`filename`*.

The form `option=val1+val2+val3` is also supported. For instance, `tabling=variant+reactive`.

These options are explained in detail in Section 23.

`?- setsemantics{`*`Option1, Option2, ...`*`}.`
This is an executable version of the `setsemantics` directive. The following options can be used only with the executable version of `setsemantics`:

`subclassing=strict` (default), `subclassing=nonstrict`,
`class_expressions=on`, `class_expressions=none` (default),

`?- setsemantics{`*`Option1, Option2, ...`*`}@`*`module`*`.`
Same as above, except that the semantics 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 FLORA-*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 knowledge base 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.

45.2 Miscellaneous Compiler Options

Sometimes it is desirable to pass miscellaneous compiler options to the FLORA-*2* compiler. To do this, FLORA-*2* provides the directive `compiler_options`. It takes one argument — a list of options that is understood by the underlying Prolog compiler. At present, the following options are supported:

- `production=on` – compile the file without various debugging features such as the rule Ids embedded in the heads and bodies of rules.
- `production=off` – compile the file in the development mode. In this mode, rule Id information is added to the heads and the bodies of the rules, which facilitates tracing and profiling of FLORA-*2* queries. This mode is the default. This compiler option was introduced to make it possible to override production mode when it is requested from the FLORA-*2* shell via the primitive `production{on}` (see Section 49.2).
- `expert=on` – allow advanced syntax. Do *not* turn this option on if you are not a very experienced FLORA-*2* user: the expert mode was introduced to prevent misuse by novice users. See Section 52 to learn about the features that are available only in the expert mode.
- `expert=off` – do not allow advanced syntax. Note that, like `production`, the expert mode is a compile-time option. In the FLORA-*2* shell, use `expert{on}` and `expert{off}`.
- `prolog=[Opt1,Opt2,...,OptN]` – pass the list of Prolog compiler options `[Opt1,Opt2,...,OptN]` to the underlying Prolog compiler. The `prolog` options are

not checked by the FLORA-2 compiler and are simply passed to the underlying Prolog engine. If any of the given options is invalid, *runtime* warnings or errors will result. Also, unlike `expert` and `production` options, the `prolog` option cannot be issued at run time via the FLORA-2 shell.

Example:

```
:- compiler_options{production=on,expert=on,prolog=[spec_repr]}.
...
?- expert{off}.
```

will cause the module that contains this directive to be compiled without the special debugging features that slow the running code down. This will also pass the `spec_repr` option to the underlying Prolog compiler so it will perform the specialization optimization.[25]

46 FLORA-2 System Modules

FLORA-2 provides a number of useful libraries that the user 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.

46.1 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.[26] The methods and predicates accessible through the `\io` library are listed below. Note that some operations are defined as transactional 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.

[25] Although the `spec_repr` optimization option is the default in XSB, FLORA-2 turns it off because, we believe, it is buggy and causes some FLORA-2 queries to go wrong.

[26] See Section 19 for a discussion of the problems associated with this representation mismatch.

46.1.1 Standard I/O Interface

This interface to input-output uses the concept of default streams that can be opened with `see` (for input) and `tell` (for output). Subsequent `see`/`tell` commands push the current streams to the respective input and output stacks and open new default streams. Subsequent reads and writes operate with those default streams. The commands `seen` and `told` close the current default streams and pop up the appropriate streams from the appropriate stacks, making them the default streams. The `seeing` and `telling` commands obtain references to the current default streams. If these streams are pushed on the stack and are no longer the default ones, those references can still be used to write to or read from the streams that are no longer default. This could be useful, for example, if one wants to copy one file to another. The following is a list of commands in this interface followed by some examples. Again, keep in mind that all the calls below must be invoked with `@\io`.

- `see(?Filename), ?Filename[see]` — open `?Filename` and make it the current input stream. The file can live remotely at some URL. In that case, `?Filename` must be instantiated to the form `url(...)`.
- `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. The file can live remotely at some URL. In that case, `?Filename` must be instantiated to the form `url(...)`. In that case, FLORA-2 will attempt to use the HTTP POST request to store the file remotely.
- `telling(?Stream)` — binds `?Stream` to the current output stream.
- `told` — closes the current output stream.
- `write(?Obj)` — writes `?Obj` to the current output stream.
- `writeln(?Obj)` — same as above, except that the newline character is output after `?Obj`.
- `write(?Obj,?Options)`, `writeln(?Obj,?Options)` — like `write(?Obj)` and `writeln(?Obj)` but takes the `?Options` argument, which is a list. At present, the only options in that list that are supported are `oid` and `goal`. The difference shows only when printing reified statements: with the `oid` option, `?Obj` is printed as an object, while with `goal` it is printed as a goal (without the reification symbol). For instance,

```
?- write(${p(1)},[oid])@\io.
```

```
${p(1)}
?- write(${p(1)},[goal])@\io.
p(1)
```

If the argument is not a reified formula then the two options give the same result. The 1-argument versions `write(?Obj)`, `writeln(?Obj)` are equivalent to `write(?Obj,[oid])`, `writeln(?Obj,[oid])`, respectively.

- `nl` — writes the newline character to the current output stream.
- `write_canonical(?Term)` — write `?Term` to standard output in canonical Prolog form.
- `fmt_write(?Format,?Term)` — C-style formatted output to the standard output. See the XSB manual, volume 2, for the description of all the details. Here we just mention that the format is an atom `'....'` whose structure is like in C (some formats might not be supported) with the addition of the format specified `%S`, which can take any term. `?Term` must have the format $args(arg_1, ..., arg_n)$ (i.e., all the arguments to be printed must be grouped under a singe functor; the name of the functor is immaterial).
- `fmt_write(?Format,?Term,?Options)` — like `fmt_write(?Format,?Term)`, but take options whose meaning is the same as in case of `write`/`writeln` above.
- `fmt_write_string(?String,?Format,?Obj)` — same as above, but binds `?String` to the result. See the XSB manual for the details.
- `fmt_write_string(?String,?Format,?Obj,?Options)` — same as above, but takes options whose meaning is the same as in the case of `write`/`writeln` above.

46.1.2 Stream-based I/O

Stream-based I/O is like the standard I/O interface except that it does not use the default streams that can be pushed to or popped from the stack of streams. Instead, there is a notion of standard input and output streams, which are usually associated with and interactive window, and user-defined streams. Standard input and output streams always exist, while user-defined streams are created and destroyed when files are open and closed. This is somewhat similar to the I/O interface described earlier, but nothing gets pushed on stacks. Instead, all read and write operations must use the appropriate streams explicitly.

For reading, we recommend to use only the stream-based I/O and so we omitted most of the read operations from the standard I/O interface above.

Note that to read an entire file via the various read operations below, it is recommended to use the fail-loop idiom like

```
%do_read(?Stream) :- ?Stream[some_read_operation(...) -> ?Result]@\io,
                     process_result(?Result), \false.
%do_read(?Stream) :- ?Stream[close]@\io.
```

where *some_read_operation* can be any of the read methods described below.

- `?Filename[open(?OpMode,?Stream)]` and `?Filename[open(?OpMode)->?Stream]`— open `?Filename` in mode `?OpMode` (which can be one of these: `read`, `write`, `append`, `write_binary`, or `append_binary`) and binds `?Stream` to the stream Id. The file can live remotely at some URL. In that case, `?Filename` must have the form `url(...)`.

  The modes `write_binary` and `append_binary` are used in Windows only. In Unix-based systems (Linux, Mac, etc.), these modes coincide with `write` and `append`, respectively.

  One way to think of `?Filename[open(read,?Stream)]` is that it is like `see(?Filename)` followed by `seeing(?Stream)`. Similarly, `?Filename[open(write,?Stream)]` is equivalent to `tell(?Filename)` followed by `telling(?Stream)`. But `open` provides one more option, `append`, which does not exist in stream-based I/O, plus the binary modes for Windows.

- `?Stream[close]` — closes `?Stream`. It is similar to `told` but it can close any stream, not just the default one.

- `?Stream[prolog_read->?Result]` — bind `?Result` to the next Prolog term in the previously open input stream `?Stream`. Each term must be terminated with a period. After being read in, each term is converted to HiLog. If the term being read is not in the form of a Prolog term, `?Term` gets bound to `prolog_read_error`.

- `?Stream[write(?Obj)]`, `?Stream[writeln(?Obj)]` — like `write(?Obj)`, `writeln(?Obj)` but write to a previously open output `?Stream` instead of the current output stream.

- `?Stream[write(?Obj,?Option)]` and `?Stream[writeln(?Obj,?Option)]` — like `write(?Obj, ?Option)` and `writeln(?Obj,?Option)` but write to a previously open output `?Stream` instead of the current output stream.

- `?Stream[fmt_write(?Format,?O)]` — similar to `fmt_write(?Format,?O)`, but uses `?Stream` for the output.

- `?Stream[fmt_write(?Format,?O,?Options)]` — similar to `fmt_write(?Format,?Term, ?Options)`, but uses a previously open stream.

- `?Stream[fmt_read(?Format) -> ?Result]` — formatted read; uses `?Stream` for input. The parameter `?Format` is a C-style format string with some limitations (and a new format `%S`, which accepts any term). See the XSB manual for details. If `?Result` is returned unbound, it means that there was a formatting read error. In that case, the current position within

the input file would not change and, if one continues to persist with exactly the same read command (e.g., using the fail-loop) then an infinite loop may result.

- `?Stream[write_canonical(?Term)]` — write out `?Term` using the canonical format. Uses `?Stream` for output.

- `?Stream[read_canonical->?Term]` — reading canonical terms (i.e., no infix, postfix, prefix operators); uses `?Stream` for input. As with `prolog_read`, each term must be terminated with a period and is converted to HiLog. If the term being read is not in the canonical form, `?Term` gets bound to `read_canonical_error`.

  The `read_canonical` method is much faster than the `prolog_read` method described earlier, but it cannot read non-canonical terms like `a+b`. So, if a file contains only

- `?Stream[flora_read->?Term]` — reads FLORA-2 terms from `?Stream`, including HiLog and reified terms. Each term must be terminated with a period. If input is not a valid FLORA-2 term, `?Term` gets bound to `flora_read_error`.

  This is similar to the `prolog_read` and `read_canonical` methods, but is more general, as this method also understands HiLog (e.g., `a(b)(c)`) and reified (e.g., `${Mary[age->12]}`) terms. The three methods have significant overlap, where `read_canonical` is least general, but also the fastest, `prolog_read` is more general, but slower, and `flora_read` is most general and also the slowest. Therefore, if the contents of a file permits so, `read_canonical` should be preferred.

  Note that this method is similar to `readAll` in the module `\parse` in Section 46.6. The difference is that it is simpler to use: it does not return any status or error information and instead binds `?Term` to `flora_read_error` in case of an error. It also ignores spaces and newlines.

- `?Stream[readline(?Type) -> ?String]` — reads a line from file; uses `?Stream` for input. Binds `?String` to the line that was just read. `?Type` is either `atom` or `charlist`. The former means that `?String` is to be bound to a Prolog atom (FLORA-2 symbol), while the latter tells the system to convert the line to a *list of characters* (usually used when further parsing is required). If the line that was read in ends with a newline, that newline character is retained in `?String`.

Here are some examples. To read a file, one must open it first. If, say, `foo.txt` has the facts `foo` and `p(a)` then the following will result:

```
?- (see('foo.txt'), seeing(?Stream),
    ?Stream[prolog_read->?X, prolog_read->?Y, seen])@\io.
```

```
?X = foo
?Y = p(a)
```

The following illustrates writing to a file.

```
?- (tell('foo.txt'), telling(?Stream1), tell('bar.txt'),
    ?Stream1[writeln(abc)], writeln(cde), told, ?Stream1[close])@\io.
```

In this case, the files `foo.txt` and `bar.txt` will be created (or erased, if they exist) and `abd` will be written out to the first file and `cde` to the second. In more detail, the first `tell` opens `foo.txt` and the next `telling` command obtains a reference to the stream associated with that file. The next `tell` opens `bar.txt`. The default stream now becomes associated with that file, but we still have a reference `?Stream1` to the first stream, so we can write to it, which is done by the next command. After that, `cde` gets written to the default stream and `told` closes that stream. The last command closes (the non-default) `?Stream1` explicitly.

To read from or write to the window through which the user interacts with a running session of FLORA-2, one does not need to open (or `see`/`tell`) any files: when FLORA-2 starts, the default input and output streams are connected to that window. For instance,

```
?- write(foobar)@\io.
```

will display `foobar` to the user unless the current default was changed by an earlier `tell` command. If the user needs to work with several files at once, he must keep track of the open streams by binding them to variables, as in the above example.

46.1.3 Controlling the Display of Floating Point Numbers

This feature controls how floating point numbers are going to be shown by the various write commands that do not themselves have the ability to express the format (which includes pretty much all the write-commands other than `fmt_write()` and the like). The floating point is controlled by three parameters:

- *Style*: `f`, `F`, `e`, `E`, `g`, and `G`. The `f`-style shows these numbers using decimal point; `e` and `E` displays them using a mantissa and an exponent (e.g., 1.23450e+01 or 1.23450E+01); and `g`, `G` use the shorter of the two and omit insignificant zeros. The default style is `g`.
- *Width*: a non-negative integer specifying the *minimum* number of characters to print.
- *Precision*: a positive integer specifying how many digits to print after the dot (for `f`, `F`, `e`, `E`) or the total number of significant digits to print overall (in case of `g` and `G`).

For more details, consult any manual page for `printf` in C or C++. (Note: Java, Python, and some other languages use slightly different semantics for printf.)

In FLORA-2, one can set the display format using this command:

```
?- setdisplayformat{float(style=...,precision=...,width=...)}.
```

Some of the specifiers above can be omitted in which case that specifier will not be changed. For instance,

```
?- setdisplayformat{float(precision=4,style=f)}.
```

To see the current settings, use `displayformat{...}`:

```
displayformat{float(?X)}.

?X = precision = '4'
?X = style = f
?X = width = '9'
```

or, for instance,

```
displayformat{float(style=?X)}.

?X = f
```

46.1.4 Display Mode and Schema

The above output methods and predicates make every effort to present the output in the valid FLORA-2 syntax. However, sometimes—usually for debugging—one would like to see the internal representation of certain FLORA-2 constructs. Using `write(...)@\prolog` is the ultimate way to see the internals, while `setdisplaymode{...}` and `setdisplayschema{...}` can be seen as higher-level controls.

There are these display modes: `default`, `stickydefault`, `answer`, `explanation`, `deepanswer`, and `debug`. At present, the only difference between `debug` and `default` is that in the `debug` mode the Skolems are written out using their internal representation while in the default mode the output looks like a FLORA-2 term. For instance,

```
? setdisplaymode{debug}.
```

```
?- insert{p(\#)}, p(?_X), writeln(?_X)@\io.
_$_$_ergo'skolem2|_2

? setdisplaymode{default}.
?- insert{p(\#)}, p(?_X), writeln(?_X)@\io.
\#
```

In the `answer` mode the values of the variables bound to Skolems or IRI prefixes are shown both in the user-readable as well as the internal representation. (Skolems and IRI prefixes that appear as *arguments* to other terms are not shown with their internal representations.) This mode is set automatically when printing out answers:

```
:- iriprefix{foo="http://foo.bar.example.com"}.
p(\#foo,f(\#2)).
?- p(?X,?Y).
?X = \#foo  (_$_$_ergo'skolem2|foo'1)
?Y = f(\#2)
?- prefix{foo,?expansion}.
?expansion = foo#''  (http://foo.bar.example.com)
```

Note that for `?Y` the internal representation of the Skolem constant is not shown because it appears as an argument, and including the internal representation would be confusing. If, however, one wishes to see internal representations of Skolem constants at any depth, setting the display mode to `deepanswer` will do the trick. To suppress showing the internal representation completely, set the display mode to `stickydefault`. The difference between this mode and `default` is that `default` temporarily changes to `answer` when answers are being printed, while in the `stickydefault` mode this automatic switch does not happen.

The `explanation` mode may be desirable when presenting explanations to the user, which is a feature in $\mathcal{ERGO}$, but may also be useful in some other special cases. The main difference between this and `default` is that in the explanation mode the names of the primitive types are not shown. For instance, in the default mode, a date would be shown as `"2018-09-19"^^\date`, while in the explanation mode it will appear simply as `2018-09-19`, which is more familiar to an end user.

One other useful display mode is `visiblechars`; it affects `write(...)@\io` predicates only. In that mode, invisible characters like backspace or newline, that have visible ASCII representation are shown using that representation. For instance,

```
?- ?_X = 'abc\bdef\n123', writeln(displayed_as=?_X)@\io.
displayed_as = 'abdef
123'
```

but

```
?- setdisplaymode{visiblechars}, ?_X = 'abc\bdef\n123', writeln(displayed_as=?_X)@\io.
displayed_as = 'abc\bdef\n123'
```

FLORA-2 also provides the primitive `displaymode{?X}`, which can be used to find out the current mode in effect. At present, only one mode can be in effect, so setting a new mode cancels the old one.

The primitive `setdisplayschema{...}` can be used to affect whether the answers and terms are printed using the FLORA-2's representation or the internal Prolog representation. For instance,

```
?- setdisplayschema{raw}.
?- ?X=p(?p).
?X = flapply(p,_h9195)
```

while in the default schema, `flora` (or `ergo`, in ERGO), the answer would be shown as

```
?-  ?X=p(?p).
?X = p(?_h9195)
```

This display schema also affects the `write(...)@\io` predicates. Another useful display schema is `tmpraw`. As the name suggests, this is a temporary raw mode that has effect only during one query. When the query is finished, the display schema returns to be `flora` or `ergo`, depending on the system. For instance,

```
?- setdisplayschema{tmpraw}, ?X=p(?).  // has effect just for this query
?X = flapply(p,_h12379)   // temporarily used the raw schema to show ?X
?- ?X=p(?).
?X = p(?_h12)       // display schema returned to be flora/ergo in next query
```

Finally, the primitive `displayschema{?X}` can be used to tell which schema is currently in effect. For instance,

```
?- displayschema{?X}.
?X = flora
```

46.1.5 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[isdir(?F)].` True if file `?F` is a directory.
- `File[isplain(?F)].` True if file `?F` is a plain file, not a directory.
- `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(?Dir)].` Makes a directory named after the value of `?Dir`.
- `File[rmdir(?Dir)].` Removes the directory `?Dir`.
- `File[chdir(?Dir)].` Changes the current directory to `?Dir`.
- `File[cwd->?Dir].` Binds `?Dir` 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`.

46.2 Storage Control

FLORA-2 keeps the facts that are part of the knowledge base or those that are inserted at runtime 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`. Transactional updates to other modules are unaffected.
- `purgedb(?Module)` — deletes all facts previously inserted into the storage associated with module `?Module`.

46.3 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 of Unix and Windows in that when the compiler or the loader are trying to locate a file specified by its name only (without directory) then they examine the files stored in the directories on the library search path.
  Using `Libpath[add(a(?Path))]` will move the directory to the front of the library search path (deleting any other occurrences of that directory on that search path).
  Using `Libpath[add(push(?Path))]` will put the directory at the front of the library search path. The other occurrences of that directory on the search path stay put.
- `Libpath[remove(?Path)]` — removes one `?Path` from the library search path.
- `Libpath[removeall(?Path)]` — removes all occurrences of `?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:

- `Method[mustDefine(?Mode)]` — affects the system behavior when stumbling upon an undefined predicate or method. This method is described separately, in Section 43.1.
- `abort(?Message)` — prints `?Message` on the 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 file must contain the include statement

  ```
  #include "flora_exceptions"
  ```

- `warning(?Message)` — prints a warning header, then message, `?Message`, and continues. Output goes to the 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)`.
- `System[type->?Info]` — returns information about the system type. `?Info`, for example, could be bound to `unix/linux/64`, `macos/darwin/64`, `windows/windows/32`, depending on the system type.

46.4 Type and Cardinality Checking

This system module of FLORA-2 provides methods for testing type and cardinality constraints of the methods defined in the FLORA-2 knowledge base. The module defines the method `check` in classes `Type` and `Cardinality` of module `\typecheck` (or, abbreviated, `\tpck`). This method is described in Section 43.3.

46.5 Data Types

The system module `\basetype` 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 39.

46.6 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,?Stat)@\parse.`
- `read(?Module)(?Code,?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.

The second form does the following: If the input term is not reified, the `?Module` parameter has no effect. If the formula is reified *and* has no explicit module (i.e., `${foo}` as opposed to `${foo@bar}`), then it will be built for the module specified in `?Module`. If `?Module` is unbound then the default module `main` is assumed.

The variable `?Stat` is bound to the status code returned by the call and has the form [*OutcomeFlag*, *EOF_flag*|*ErrorList*], where:

OutcomeFlag = null/ok/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 numbers, which are largely irrelevant in this context.
Message is an error message. Error messages are displayed.

Example:

```
?- read(?Code,?Stat)@\parse.
f(a).   <-----user input

?Code = f(a)
?Stat = [ok, not_eof]
```

```
?- read(?Code,?Stat)@\parse.
${a[b->c]@foo}.   <-----user input

?Code = ${a[b -> c]@foo}
?Stat = [ok, not_eof]
```

Note that `read@\parse` reads just one term from the input and succeeds. If called repeatedly, it will read the second, third, etc., term from the input stream. On reaching the end of file, `?Stat` will be bound to `[null,eof]`.

- `?Stream[read(?Code,?Stat)]@\parse.`
- `?Stream[read(?Module)(?Code,?Stat)]@\parse.`

These versions of `read@\parse` are similar to the above except that the input comes from an input stream `?Stream`, which must be open previously, and not from the standard input.

The second form of this API call supplies a module for reified terms, as explained above.

- `readAll(?Code,?Stat)@\parse.`
- `readAll(?Module)(?Code,?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).

The second form of this API call supplies a module for reified terms, as explained above.

- `?Stream[readAll(?Code,?Stat)]@\parse.`
- `?Stream[readAll(?Module)(?Code,?Stat)]@\parse.`

Like `readAll(?Code,?Stat)@\parse` but the input comes from the input stream `?Stream`, which must be open in advance. After finishing working with the stream, it should be closed.

The second form of this API call provides a module for reified terms, as above.

Note: there is a simpler version of this method in the system module `\io`. It is called `flora_read` (and `flora_read(?Module)`) and is described in Section 46.1.

- `?Source[readAll(?CodeList)]@\parse.`
- `?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). `?CodeList` gets 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.

The second form of this call provides an explicit module for building reified terms, as above.

46.7 Displaying FLORA-2 Terms and Goals

The system library loaded into the module `\show` can be used to obtain printable representation of FLORA-2 terms and goals. This is needed when one needs to show those terms to the user in a form that the user can recognize (rather than in the internal form into which the terms are compiled).

The available methods are:

- `?Term[show->?Result]` — `?Result` is bound to an atom that represents the printable view of the term. For instance,

```
?- ${d(c,k,?M)}[show->?P]@\show.

?P = '${d(c,k,?_h0)@main}'

?- [f(a,?X),b,${d(c,k,?M)}][show->?Result]@\show.

?P = '[f(a,?_h0), b, ${d(c,k,?_h1)@main}]'
```

- `?Term[show(?Option)->?Result]` — like the above but also takes an option argument, which can be `goal` or `oid`. `?Term[show->?Result]` is the same as `?Term[show(oid)->?Result]`. The `goal` option affects the display of reified statements only. In that case, these statements are shown as goals (without the reification symbol), while with `oid` they are shown as objects. For instance,

```
?- ${p(a,b)}[show(oid) -> ?R]@\show.
?R = '${p(a,b)@main}'

?- ${p(a,b)}[show(goal) -> ?R]@\show.
?R = 'p(a,b)'
```

For HiLog terms, which are not reified, the two forms give the same result. For instance,

```
?- p(a,b)[show(oid) -> ?R]@\show.
?R = 'p(a,b)'
```

- `?List[splice(?Separator)->?Result]` — `?Result` is bound to an atom that represents the printable view of the sequence of elements in `?List` with `?Separator` (an atom) inserted in-between every pair of consecutive elements. For instance,

```
?- [f(a,?X),b,${d(c,k,?M)}][splice(' ')->?P]@\show.

?P = 'f(a,?_h0) b ${d(c,k,?_h1)@main}'

?- [f(a,?X),b,${d(c,k,?M)}][splice('| |')->?P]@\show.

?P = 'f(a,?_h0)| |b| |${d(c,k,?_h1)@main}'
```

47 Unicode and Character Encodings

If all your applications deal only with ASCII characters, like the ones found on an English keyboard, you can safely skip this section. Otherwise, if your are dealing with alphabets other than English (even if it is just one of the European languages based on Latin alphabet) then read on.

47.1 What Is a Character Encoding?

Data (including programs, databases, and knowledge bases) is represented as sequences of characters, where each character is encoded as a sequence of bits. All these bits are useless, however, unless one knows what character each particular subsequence represents. The mapping between bit sequences and characters is known as a *code table*. The best-known coding table is *ASCII*, and it assigns a number between 0 and 127 to various characters found on English keyboards, including digits, lower- and uppercase letters, and punctuation.

Unfortunately, ASCII encodes too few characters and various groups took initiative to define bigger code tables, typically appropriating the characters in the 128 – 255 range for various national alphabets. One of the best-known such tables is *Latin-1,* which encodes all special accented letters in Latin-based alphabets. But these extra 128 symbols are too few to accommodate other alphabets, like Russian, Greek, Hebrew, etc. As a result, many more coding tables sprang up—all appropriating the same characters from the range 128 – 255 for the different alphabets, and this

is where many problems originate. Data could now come in any of the dozens of different encodings and, unless one knows which encoding was used, that data is all but useless. On top of all this, Far-Eastern languages need so many characters for writing that they require different kinds of coding tables all together.

To bring some order into this business, *Unicode Consortium* was created and tasked to come up with *one* coding table that is capable of encoding all existing alphabets and common characters, including many mathematical characters, currency symbols, and then some. This eventually led to the Unicode standard known as UTF-8, which is a coding table that represents all known characters using sequences of one to four bytes. Thus, ASCII characters are represented using one byte, Latin-1 (and much more) using two bytes, etc. This picture is complicated by the fact that Unicode Consortium did not come to UTF-8 right away, but through a series of less successful standards, like UTF-16 and UTF-32, which are still around and make things more complicated.

47.2 Character Encodings in $\mathcal{F}$LORA-*2*

With all this multitude of encodings, $\mathcal{F}$LORA-*2* supports only the three most common ones: UTF-8, CP1252, and Latin-1. Since these three (and, in fact, all coding systems) subsume and agree on the ASCII character set, the latter is, of course, also supported.

CP1252 is very similar to Latin-1 in its intent and their encodings agree everywhere except for about two dozen of symbols in the 128 – 159 range. The only reason why CP1252 is supported by $\mathcal{F}$LORA-*2* is that it happens to be the default on Windows and much of the data produced on that platform still tends to be encoded using the CP1252 coding table. However, in $\mathcal{F}$LORA-*2* itself, the default encoding is UTF-8 in Linux, Windows, and Mac.

47.3 Specifying Encodings in $\mathcal{F}$LORA-*2*

To specify the encoding to use in processing programs or data, $\mathcal{F}$LORA-*2* provides one compiler directive and one executable directive as follows:

```
:- encoding{enc_name}.         // compiler directive for FLORA-2 source files
?- encoding{stream,enc_name}.  // executable directive for data files
```

where *enc_name* can be one of the following: `utf8`, `latin1`, `cp1252`; *stream* is an open file handle. The use of these directives is explained next.

First, if a rule base or a data file contains only ASCII characters, the encoding is immaterial and nothing needs to be specified. Otherwise, if one has a ruleset or a set of $\mathcal{F}$LORA-*2* facts that contain non-ASCII symbols, then the *compiler directive* `encoding` might need to be used at the

top of the corresponding file. If a data file that contains non-ASCII characters needs to be read in or written out, the *executable directive* might need to be executed just *after* the file is opened.

FLORA-*2* uses UTF-8 as its default encoding, so if a rule base or a data file have this encoding then nothing needs to be done. If a rule base has a different encoding (`latin1` or `cp1252`) then the appropriate compiler directive needs to be placed at the top of the file (e.g., `:- encoding{cp1252}.`). If a data file has an encoding other than ASCII or UTF-8, an appropriate executable directive should be executed right after the file is open (e.g., `encoding{`*stream*`,latin1}.`). Files in FLORA-*2* are open using `see(...)@\io` (input), `tell(...)@\io` (output), or `File[open(?mode)->?stream]@\io` (both input and output) operations—see Section 46.1. The *stream* argument in the encoding directive is an identifier of the file in question. For files open with the `see` command, this identifier can be obtained by calling `seeing(?Stream)@\io`; for files open with the `tell` command, the stream identifier is obtained by calling `telling(?Stream)@\io`. If a file was open using the `open` method, the stream identifier is returned by that method (see Section 46.1).

Note that the *scope* of the compiler directive `encoding` is the file in which the directive occurs (and all the `#include`'d files). After the file is compiled, the encoding returns to what it was before. The scope of the *executable* encoding directive is the open file to which the directive applies.

48 Notes on Style and Common Pitfalls

Knowledge engineering in FLORA-*2* is similar to programming in Prolog, but is more declarative. For one thing, frame literals are always tabled, so the knowledge engineer 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 knowledge base are stored in special data structures so that their order does not matter and duplicates are eliminated automatically.

48.1 Facts are Unordered

The fact that FLORA-*2* does not assume any particular order for facts has a far-reaching impact on the knowledge engineering style and is one of the pitfalls that an engineer 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 idiom is usually used when the "..." contains some procedural statement, like writing on the screen, but it will *not* work in FLORA-*2*, because `p(?_)` will be treated as a database fact, which is placed in no particular order with respect to the program. If one wants the same effect in FLORA-*2*, 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.
```

Here we added %-sign to `p` because, as noted above, this kind of an idiom is used when "..." contains some procedural statement.

48.2 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 FLORA-*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-based knowledge engineering 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, ?X[building->?B] :- ?X[department->?[building->?B]].
```

A minute of reflection should convince the reader that this 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 carefully check the validity of placing class membership formulas in rule heads.

48.3 Composite Frames in Rule Heads

Another common mistake is the inappropriate use of complex frames in rule heads. When using a complex frame, 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 frame occurs in a 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 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, `flightobj` 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 frames 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 a rule head. The error in the above is 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, f(?F,?T)[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 `skolem{...}` primitive from Section 14. Here we are using the fact that each time `flight(?F,?T)` is satisfied, `skolem{?X}` generates a new value for `?X`.

```
%convert_rel_to_oo :-
     flight(?F,?T), skolem{?O}, insert{?O:flight[from->?F,to->?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.

49 Miscellaneous Features

49.1 Suppression of Banners

When *FLORA-2* initializes itself, it generates quite a bit of chatter, which is suppressed by default. The user who needs this information (e.g., the developer), can force the chatter to appear by starting *FLORA-2* with

```
runflora --devel
```

In normal operation, FLORA-2 issues a prompt after every query or command. However, sometimes it might be necessary to suppress the prompt. 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 --noprompt
```

will print no chatter, not even the prompt, 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 input without issuing any prompts.

49.2 Production and Development Compilation Modes

By default, FLORA-2 compiler compiles everything in the development mode. However, before deploying an application, it is desirable to recompile it in the production mode to gain significant performance benefits. One way to do this is to put the directive

```
:- compiler_options{production=on}.
```

in each file. However, this is often inconvenient, takes time, and might be error-prone. An alternative method is to execute the following command at the prompt:

```
?- production{on}.
```

Executing

```
?- production{off}.
```

puts FLORA-2 back into the development mode.

Note that the mode is reset to the default development mode after compiling any file, so `production{on}` must be re-executed each time when compilation in that mode is desired.

Also note that the explicit compiler directives

```
:- compiler_options{production=on}.
:- compiler_options{production=off}.
```

placed at the top of a file override any prior `production{on}/production{off}` commands.

50 Useful XSB Predicates Without a Counterpart in FLORA-2

This section contains a list of useful predicates that are available in XSB, have no direct counterpart in FLORA-2, and for which such counterparts are not being planned.

50.1 Time-related Predicates

- `cputime(?X)@\prolog` — returns the CPU time in seconds (floating point number; including fractions of seconds) used by the FLORA-2 process so far. This excludes the idle time.
- `walltime(?X)@\prolog` — returns the total time in (including fractions of seconds) seconds that elapsed since FLORA-2 started, including the idle time.
- `epoch_seconds(?X)@\prolog(machine)` — time in seconds (no fractions) that has passed since Thursday, 1 January 1970.
- `epoch_milliseconds(?Secs,?Fraction)@\prolog(machine)` — time that has passed since Thursday, 1 January 1970. Both `?Secs` and `?Fraction` are integers, where `?Secs` is the number of seconds and `?Fraction` is the number of additional milliseconds since the above date.

50.2 Hashing

- `term_hash(?Term, ?Size, ?Value)@\prolog(machine)` — binds `?Value` to the hash number of `?Term` in the range from 0 to `?Size-1`.
- `crypto_hash(?Type, ?Input, ?Result)@\prolog(machine)` — produce a cryptographic hash of the input and bind `?Result` to it. Input must be either an atom or have the form of `file(filename)@\prolog`. `?Type` specifies the type of the hash function to use. Currently, only `md5` (for MD5 hash) and `sha1` (for SHA1) are supported.

50.3 Input/Output

- `write(?X)@\prolog` and `writeln(?X)@\prolog` — these are similar to the corresponding predicates in the FLORA-2 module `\io`, but they print out the internal form of the term bound to `?X`. For atoms, numbers, and variables, there is no substantial difference with `\io`, but for more complex terms there is. These predicates are mostly useful for debugging FLORA-2 itself and for bug reporting.

50.4 Meta-programming

FLORA-2 has a high-level predicate `=..`, which lets one inspect the internal structure of most FLORA-2's constructs in high-level terms of that system. However, if more sophisticated, low-level parsing of terms is needed, the following Prolog predicates can be used:

- `(?Left =.. ?Right)@\prolog` — decomposes the `?Left` term into a list of the form `[functor, arg1,...,argN]`. The result is bound to `?Right`.
- `functor(?Term,?Functor,?Arity)@\prolog` — binds `?Functor` to the function symbol used in `?Term` and `?Arity` to the arity of `?Term`.
- `arg(?ArgNum,?Term,?Arg)@\prolog` — binds `?Arg` to the `?ArgNum`'s argument of `?Term`.

51 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 knowledge base. A compiler or a runtime error issued for a perfectly valid FLORA-2 specification is probably a bug in FLORA-2.

Bugs that are the fault of the underlying Prolog engine are particularly hard to fix, because FLORA-2 knowledge bases 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{...}` primitive can be used to strip the macros from the code, making it much easier to understand. If you issue the following command

```
?- dump{foo}.
```

then `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,

```
?- 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 specification 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 FLORA-2, 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 knowledge base 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 knowledge base has several user modules, try to put them into one file and use just one module.
4. Use `dump{...}` to strip FLORA-2 macros from the output of the FLORA-2 compiler.
5. See if the resulting program runs under XSB (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 rules and squeezing a multi-module program into just one file.
2. Remove all the calls to system modules, unless such a call 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 queries may cause the following XSB error message:

   ++Error[XSB]: [Compiler] '!' after table dependent symbol

   or something like that. This is due to certain limitations in the implementation of tabled predicates in the XSB system. This problem might be eliminated in a future release of XSB.
2. Inheritance of transactional methods is not supported: `a[|%p(?X)|]`.

52 The Expert Mode

Skip this section unless you are an experienced *FLORA-2* user who has good understanding of the syntactic, semantic, and computational aspects of *FLORA-2*.

FLORA-2 has plethora of syntactic constructs, which may be daunting for a novice and certain constructs are especially prone to be misused by such users to detrimental effect. To fence off

these features from a novice, FLORA-2's parser works in the novice mode by default. Only very experienced users should work in the expert mode—see Section 45.2 for how to do that.

The following syntactic constructs are available only in the expert mode:

- *Shortcut for charlists.* This shortcut permits to write `"abc"` instead of `"abc"^^\charlist`. Charlists are commonly used for parsing, but novice users tend to misuse them for the same purpose as atoms. Because charlists are much more expensive in terms of memory, compilation time, and various runtime operations, using them in place of atoms is a very bad idea.

- *Embedded ISA-literals in rule heads.* An embedded ISA-literal is one that appears as an argument to a predicate, function symbol, or is part of a composite frame. For instance, `p(a:b)`, `p(a,f(c::d))`, `a:b[c->d:f]`. FLORA-2 allows these to appear in rule bodies and in facts, but their appearance in rule heads (that have non-empty body) is restricted to the expert mode.

  One reason why embedded ISA-literals in rule heads are not advisable to novice users is because they tend to confuse them with typed variables. Note that

  ```
  p(?X:foo) :- ...
  p(?X^^foo) :- ...
  ```

  are very different things and using the first form is frequently a mistake. An embedded ISA-literal, `p(?X:foo)`, represents the conjunction `p(?X), ?X:foo`, so the first rule above is equivalent to

  ```
  p(?X), ?X:foo :- ...
  ```

  That is, `?X:foo` is *derived*, not checked. Inexperienced users tend to incorrectly assume the latter. On the other hand, the second rule, one that uses typed variables is equivalent to

  ```
  p(?X) :- ?X:foo, ...
  ```

  but using `?X^^foo` can be much more efficient, since the actual tests for class membership are done only when the variable gets bound.

- *Expanded scope of the operators* `->` *and* `=>`. The symbols `->` and `=>` are infix operators when they appear in the context of a frame. In other contexts, using them as infix operators will cause an error, as mistyping round parentheses for square brackets is a common mistake. Advanced users, however, may wish to use these operators for other purposes as well, e.g., to simulate predicates with named arguments, such as `p(foo->1,bar->2)`. However, this syntax, is allowed only in the expert mode.

- *The* `<==>` *and* `<~~>` *double implications.* These double-implications are blocked off in the novice mode because inexperienced users tend make logical mistakes by using “iff” when only “if” is called for.

Appendices

A A BNF-style Grammar for FLORA-2

This BNF is an approximation of the operator-based, context sensitive syntax used in FLORA-2. Not all features mentioned in the preceding sections (especially directives and a number of non-logical commands) are listed in this BNF.

```
%% 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-2 these symbols can be used with or without the quotes.

Statement := (Rule | Query | LatentQuery | Directive) '.'

Rule := (RuleDescriptor)? Head (':-' Body)? '.'

Query := '?-' Body '.'

LatentQuery := RuleDescriptor '!-' Body '.'

Directive := ':-' ExportDirective | OperatorDirective | SetSemanticsDirective
             | IgnoreDependencyCheckDirective | ImportModuleDirective
                     | PrefixDirective | CompilerDirective | IndexDirective

RuleDescriptor := '{' RuleTag '}'
                  | '{' BooleanRuleDescriptor '}'
                  | '@!{' RuleId ( '[' DescrBody ']' )? '}'

RuleTag := Term
RuleId := Term
BooleanRuleDescriptor := Term
DescrBody := DescrBodyElement (',' DescrBodyElement)*
DescrBodyElement := Term | Term '->' Term

%% Heads in ERGO (not FLORA-2) can also have ==>, <==, <==>, \or, and quantifiers
Head := ('\neg')? HeadLiteral
```

```
Head := Head (',' | '\and') Head

HeadLiteral := BinaryRelationship | ObjectSpecification | Term

Body := BodyLiteral
Body := BodyConjunct | BodyDisjunct | BodyNegative | ControlFlowStatement
%% 'exists' can be used instead of 'exist'.
%% Body-parentheses are optional, if Body is BodyLiteral
Body := 'forall(' VarList ')' '^' '(' Body ')' | 'exist(' VarList ')' '^' '(' Body ')'
Body := Body '@' ModuleName
Body := BodyConstraint

ModuleName := Atom | 'Atom()' | Atom '(' Atom ')' | ThisModuleName

BodyConjunct := Body (',' | '\and') Body
BodyDisjunct := Body (';' | '\or') Body
BodyNegative := (('\naf' | '\neg | '\+') Body)
BodyConstraint := '{' CLPR-style constraint '}'

ControlFlowStatement  := IfThenElse | UnlessDo
                       | WhileDo | WhileLoop
                       | DoUntil | LoopUntil
IfThenElse := '\if' Body '\then' Body  ('\else' Body)?
              | Body '<~~' Body | Body '~~>' Body | Body '<~~>' Body
              | Body '<==' Body | Body '==>' 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 | RuleUpdate | Refresh
               | NewSkolemOp | Builtin | Loading | Compiling
               | CatchExpr | ThrowExpr | TruthTest

Builtin := ArithmeticComparison | Unification | MetaUnification | ...

Loading := '[' LoadingCommand (',' LoadingCommand)* ']'
           | 'load{' LoadingCommand (',' LoadingCommand)* '}'
```

```
LoadingCommand := Filename ('>>' Atom)

Compiling := 'compile{' Filename '}'

BinaryRelationship := PathExpression ':' PathExpression
BinaryRelationship := PathExpression '::' PathExpression

ObjectSpecification := PathExpression '[' SpecBody ']'

SpecBody := ('\naf')? MethodSpecification
SpecBody := ('\neg')? ExplicitlyNegatableMethodSpecification
SpecBody := SpecBody ',' SpecBody
SpecBody := SpecBody ';' SpecBody

MethodSpecification :=  ('%')? Term
MethodSpecification :=  PathExpression
                        (ValueReferenceConnective | SigReferenceConnective)
                        PathExpression

ValueReferenceConnective :=  '->' | '+>>' | '->->' | '-->>'

SigReferenceConnective := ('{' (Integer|Variable) ':' (Integer|Variable) '}')? ('=>' )

ExplicitlyNegatableMethodSpecification := Term
ExplicitlyNegatableMethodSpecification :=
            PathExpression ExplicitlyNegatableReferenceConnective PathExpression

ExplicitlyNegatableReferenceConnective := '->' | SigReferenceConnective

PathExpression := Atom | Number | String | Iri | Variable | SpecialOidToken
PathExpression := Term | List | ReifiedFormula
PathExpression := PathExpression  PathExpressionConnective PathExpression
PathExpression := BinaryRelationship
PathExpression := ObjectSpecification
PathExpression := Aggregate

Iri := SQname | FullIri

SQname := Identifier '#' String
```

```
FullIri := 'Śtring

PathExpressionConnective :=  '.' | '!'

SpecialOidToken :=  AnonymousSkolem | NumberedSkolem | NamedSkolem | ThisModuleName

ReifiedFormula := '${' (Body | '(' Rule ')') '}'

%% No quotes are allowed in the following quasi-constants!
%% No space allowed between \# and Integer and \# and AlphanumAtom
%% AnonymousSkolem & NumberedSkolem can occur only in rule head
%% or in reified formulas
AnonymousSkolem   := '\#'
%% No space between \# and Integer
NumberedSkolem    := '\#'Integer
%% No space between \# and AlphanumAtom
NamedSkolem    := '\#'AlphanumAtom
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' | 'setof' | 'bagof'
%% 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 setof/bagof, use the following idiom:
%%      S = setof{ V | ... , V=Term }
TargetVariable := Variable
GroupingVariables := '[' VarList ']'

Variable := '?' ([_a-zA-Z][_a-zA-Z0-9]*)?
VarList := Variable, (',' Variable)*
```

```
DBUpdate := DBOp '{' UpdateList ('|' Body)? '}'
DBOp := 'insert' | 'insertall' | 'delete' | 'deleteall' | 'erase' | 'eraseall'
%% In ERGO, UpdateList can also contain stealth literals
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 ')' )*

NewSkolemOp := 'skolem{' Variable '}'

CatchExpr := 'catch{' Body, Term, Body '}'
ThrowExpr := 'throw{' Term '}'
TruthTest := 'true{' Body '}' | 'undefined{' Body '}' | 'false{' Body '}'
               | 'truthvalue{' Variable '}'
```

B The FLORA-2 Tracing Debugger

The 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 execution step by step, and spying allows one to tell FLORA-2 that it must pause 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 in FLORA-2.

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 a program. In that 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 parts 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 knowledge base 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 captures some of the most common uses of the aforesaid `debug_ctl` interface. Executing

```
?- \trace('foobar.txt').
```

will switch FLORA-2 to non-interactive trace mode and the entire trace will be dumped to file `foobar.txt`. Note that you have to execute `\notrace` or exit Prolog in order for the entire file to be flushed to disk.

Another useful form of non-interactive tracing is to dump the trace into a file in the form of FLORA-2 facts, so that the file could later be loaded and queried. This is accomplished with the following call:

```
?- \trace('foobar.txt',log).
```

The second argument denotes the option to be passed to the trace facility. Currently the only available option is `log`. The form of the facts is as follows:

```
flora_tracelog(CallId,CallNumber,PortType,CurrentCall,Time)
```

Here `CallId` is an identifier generated when the engine encounters a new top-level call. This identifier remains the same for all subgoals called while tracing that top-level call. `CallNumber` is the call number that the underlying generates to show the nesting of the calls being traced. It

is the same number that the user sees when tracing interactively. `PortType` is `'Call'`, `'Redo'`, `'Exit'`, or `'Fail'`. `CurrentCall` is the call being traced. `Time` is the CPU time it took to execute `CurrentCall`. On `'Call'` and `'Redo'`, `Time` is always 0 — it has a meaningful value only for the `'Exit'` and `'Fail'` log entries.

Low-level tracing. The FLORA-2 debugger also supports low-level tracing via the shell command `\tracelow`. With normal tracing, the debugger converts low-level subgoals to FLORA-2 syntax 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:

```
?- \tracelow('foobar.txt').
```

As with the `\trace/1` call, there is a `\tracelow/2` version:

```
?- \tracelow('foobar.txt',log).
```

which dumps the trace in the form of queriable facts. However, in this case the facts are in the low-level Prolog form, not FLORA-2 form.

C For Emacs Aficionados: Editing and Invoking FLORA-2 in Emacs

For power-users who prefer Emacs to all other editors, FLORA-2 includes a special major mode, *flora-mode*. This mode provides support for syntactic highlighting, automatic indentation, and the ability to load knowledge bases and pose FLORA-2 queries right out of the Emacs buffer.

Note that this mode has *not* been tested in XEmacs—only in Emacs.

C.1 Installation of flora-mode

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

```
.../flora2/emacs/flora.elc
```

found in your FLORA-2 distribution on the load path of Emacs. The best way to work with Emacs is to make a separate directory for Emacs libraries (if you do not already have one), and put `flora.elc` there. Such a directory can be added to the emacs search path by putting the following command in the file ∼`/.emacs` (in Windows the `.emacs` file will likely be in your home directory):

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

Finally, you must tell Emacs how to recognize FLORA-2 files, so Emacs will be able to invoke `flora-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-2 knowledge bases." t)
```

and where the FLORA-2 startup script is:

```
(setq flora-program-path "full-path-to-your-runflora-script")
```

This script is found in the FLORA-2 reasoner's installation folder. In Linux and Mac it is called `runflora` and in Windows it is `runflora.bat`.

C.2 Functionality of flora-mode

The *Flora-2* menu. Once FLORA-2 editing mode is installed, it provides a number of functions. First, whenever you edit a FLORA-2 file, you will see the "*Flora-2*" menu in the menubar. This

menu provides commands for starting and stopping the FLORA-2 process (i.e., the FLORA-2 shell). When this process starts, a command window will appear in a separate Emacs buffer, and you can type commands to it as in a regular terminal window. In addition, the menu *Flora-2* in the menubar lets one send queries to the FLORA-2 process directly from Emacs buffers containing FLORA-2 statements, and one can load and add portions of these buffers, entire buffers, or other files.

Because Emacs provides automatic file completion and allows you to edit what you typed, performing these operations directly out of the Emacs buffer is much faster than typing the corresponding commands to the FLORA-2 shell.

Keystrokes. 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 buffer:              Ctl-c Ctl-b
Load region:              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-2 Process:      Ctl-c Ctl-c
Quit Flora-2 Process:           Ctl-c Ctl-d
Restart Flora-2 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 the FLORA-2 shell running.

Indentation. The Emacs editing mode for FLORA-2 understands many aspects of the FLORA-2 syntax, which enables it to provide correct indentation in most cases. The only area where FLORA-2 sometimes gets indentation wrong is when nested control constructs are used (e.g., nested \if-\then-\else, \while-\do, etc.).

The most common way of using the indentation facility is by typing the `TAB`-key. This tells *flora-mode* to indent the line according to its taste. In most cases, after you finish the previous line

and type <Enter>, *flora-mode* will guess correctly where the next line should start. If it cannot guess, typing `TAB` after the line is finished will indent the line properly. Closing parentheses and some other characters are *electric*, meaning that typing them may also cause Emacs to indent the current line.

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

```
?- compile{mykb}.
```

compiles the knowledge base found in the file `mykb.flr` and generates the following files: `mykb.P`, `mykb_main.xwam`, and `mykb.fdb` (if `mykb.flr` contains facts in addition to rules). By default, `load{mykb}` loads the knowledge base into the default user module named `main`. If `mykb.flr` contains F-logic facts, all these facts will be compiled separately into the file `mykb.fdb` that is dynamically loaded at runtime. Next, the file `mykb.P` is generated — take a look at "`mykb_main.P`" to see what has become of your FLORA-2 knowledge base(!) — and passed to the Prolog compiler, yielding Prolog byte code `mykb.xwam`, which is then renamed to `mykb_main.xwam`. This file is then loaded and executed. If `mykb.flr` contains queries, they are immediately executed by Prolog (provided there are no errors).

In the module system of FLORA-2, any knowledge base can be loaded into any user module. The same file can even be loaded into two different modules at the same time, in which case there will be two distinct copies of the same knowledge base running at the same time. For each user module, a different byte code is generated (this is why `mykb.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 file `default.flr` from the demo directory by launching the command `?- demo{default}.`, pose the following query and see what happens.

```
?- ?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 [10]. It consists of two parts: translation of frames 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 frame, which represents some facts about the object `Mary`:

```
Mary:employee[age->29, kids->{Tim,Leo}, salary(1998)->100000].
```

As described in [10], any complex frame can be decomposed into a conjunction of simpler atomic F-logic formulas. These latter frames can be directly represented using Prolog syntax. For different kinds of F-logic frames we use different Prolog predicates. For instance, the result of translating the above frame might be:

```
isa(Mary,employee). // Mary:employee.
mvd(Mary,age,29). // Mary[age->29].
mvd(Mary,kids,Tim). // Mary[kids->{Tim}].
mvd(Mary,kids,Leo). // Mary[kids->{Leo}].
mvd(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` and `sub` encode the IS-A and subclass relationships, respectively. We call these predicates *wrapper predicates*. Of course, FLORA-2 has much more: signatures, class-level vss. object-level statements, directives, and all kinds of auxiliary predicates needed to improve efficiency.

FLORA-2 facts, such as those above, are then stored in a special data structure, called a trie, and are retrieved using "patch rules", which have the form

```
mvd(Obj,Meth,Val) :- storage_find_fact(TrieName, mvd(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 knowledge bases are loaded into modules dynamically, the name of the storage trie is determined at the 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 pre-pended to them and thus are different for different modules. Moreover, since module names are determined at the 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_mvd(Mary,parent,?X) :- d_mvd(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 (`\naf`, see Section 20). The prefix used in the body of a rule, `d_`, is introduced in order to be able to capture undefined methods, i.e., methods whose definition was not supplied by the user (see Section 43.1). 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_` predicates. 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 the rules:

```
derived_mvd <-- d_mvd
```

In auxiliary runtime libraries:

```
d_mvd <-- mvd
d_imvd <-- imvd
mvd <-- inferred_mvd
mvd <-- \naf inferred_mvd <-- derived_mvd
mvd <-- \naf conflict_obj_imvd <-- imvddef <-- mvd
mvd <-- imvd
mvd <-- immediate_isa

imvd <-- immediate_sub
imvd <-- inferred_imvd <-- derived_imvd
imvd <-- \naf inferred_imvd
imvd <-- \naf conflict_imvd <-- imvddef

imvddef <-- imvd

derived_mvd <-- storage_find_fact(...trie_name..., mvd(...))
```

Here we have listed only the predicates that are used to model value-returning class-level (`imvd`) and object-level (`mvd`) statements. 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`, and `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}`, the existing clauses are not overridden 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 rule heads of the clauses generated by the compiler from the clauses in the original user knowledge base, while the latter appear only in 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 knowledge bases are 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 invoked by user knowledge bases and so there is no place where one can include the trailer. Second, the user may enter the following executable instruction

```
?- setsemantics{equality=...}.
```

at the shell prompt and knowledge base may also contain these instructions as part of their content. 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 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 currently loaded knowledge bases.

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 knowledge bases, making it possible to split the specifications into separate files and import objects defined in other modules. Second, it allows FLORA-2 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 [17].

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 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.[27]

Apart from the general prefix, each predicate name's prefix contains the module name where this predicate is defined. Since the same F-logic knowledge base 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, `mykb.P`, which is generated 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
```

[27] 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.

Gpp then includes all the necessary files, and then pipes the result to the Prolog compiler. The latter produces the object `mykb.xwam` file where all the predicate names are wrapped with the user module name, as described above. This object file is renamed to `mykb_main.xwam`. If later we need to compile `mykb.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 an object file, the file is renames to `mykb_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 pre-pended with the module name. In particular, this implies that HiLog atomic formulas have different representations 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 [4].

The updatable part of the database. All objects and facts that are explicitly inserted by the user knowledge base are kept in the special *storage trie* associated with the user module where the knowledge base is loaded. A trie is a special data structure, which is well-suited for indexing tree-structured objects, like Prolog terms. This workhorse does much of the grunge 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 the predicates that represent frames, 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 the compiled code, the patch rules are included in 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 input 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 specification, 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 *canonical term* list, which represents the entire parsed specification. The canonical 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 specification 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 4 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.
- `flrutils.P`: Miscellaneous utility predicates for loading knowledge bases, checking if files exist, whether they need to be recompiled, etc.

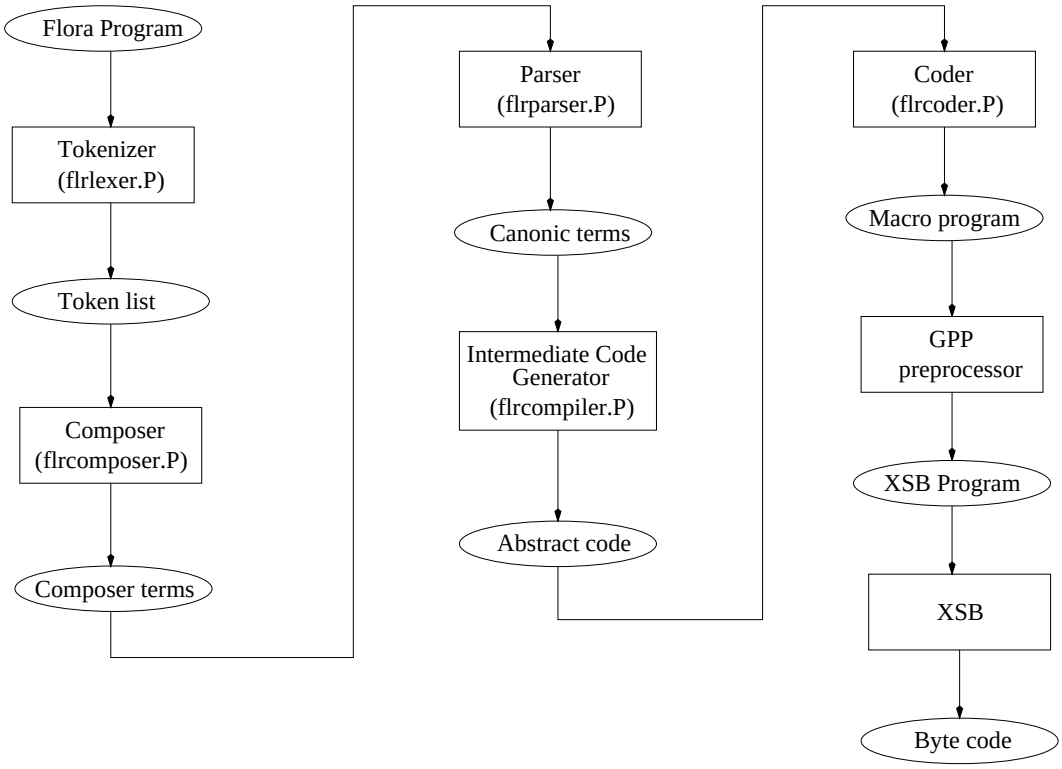

Figure 2: The architecture of the FLORA-2 system.

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 automatically determines which of these libraries are needed. 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 installation.

The directory `includes/` contains (among others) the header file, which defines a number of important macros (*e.g.*, `FLORA_THIS_WORKSPACE`) that wrap all the names with prefixes to separate the different modules found in the user knowledge base. 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 `cc` contains several C programs used by FLORA-*2*. Finally, the `pkgs/` directory contains a number of FLORA-*2* packages that are not core part of the system.

References

[1] A.J. Bonner and M. Kifer. Transaction logic programming. In *International Conference on Logic Programming*, pages 257–282, Budapest, Hungary, June 1993. MIT Press.

[2] A.J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.

[3] 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.

[4] 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.

[5] 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.

[6] K. Clark. Negation as failure. *Logic and Databases*, pages 293–322, 1978.

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

[8] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[9] 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.

[10] 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.

[11] Senlin Liang and Michael Kifer. A practical analysis of non-termination in large logic programs. *Theory and Practice of Logic Programming*, 13:705–719, September 2013.

[12] Senlin Liang and Michael Kifer. Terminyzer: An automatic non-termination analyzer for large logic programs. In *Practical Aspects of Declarative Languages (PADL)*, January 2013.

[13] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

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

[15] 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.

[16] H. Wan, B. Grosof, M. Kifer, P. Fodor, and S. Liang. Logic programming with defaults and argumentation theories. In *International Conference on Logic Programming*, July 2009.

[17] 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.

[18] 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.

[19] 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.

[20] G. Yang and M. Kifer. Inheritance in rule-based frame systems: Semantics and inference. *Journal on Data Semantics*, 2800:69–97, 2003.

[21] G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the Semantic Web. *Journal on Data Semantics, LNCS 2800*, 1:69–98, September 2003.

Index