A Guide to ℰRGO Packages

Version 1.2 (Solon)

Edited by

Michael Kifer
Coherent Knowledge

May 2017

Contents

Chapter 1

JAVA-to-$\mathcal{E}$RGO Interfaces

by Aditi Pandit and Michael Kifer

This chapter documents the API for accessing $\mathcal{E}$RGO from Java programs. The API has two versions: a *low-level API*, which enables Java programs to send arbitrary queries to $\mathcal{E}$RGO and get results, and a *high-level API*, which is more limited, but is easier to use. The high-level API establishes a correspondence between Java classes and $\mathcal{E}$RGO classes, which enables manipulation of $\mathcal{E}$RGO classes by executing appropriate methods on the corresponding Java classes. Both interfaces rely on the Java-XSB interface, called *Interprolog*, developed by `Interprolog.com`.

The API assumes that a Java program is started first and then it invokes XSB/$\mathcal{E}$RGO as a subprocess. The XSB/$\mathcal{E}$RGO side is passive: it only responds to the queries sent by the Java side. Queries can be anything that is accepted at the $\mathcal{E}$RGO shell prompt: queries, insert/delete commands, control switches, etc., are all fine. One thing to remember is that the backslash is used in Java as an escape symbol and in $\mathcal{E}$RGO as a prefix of the builtin operators and commands. Therefore, each backslash must be escaped with another backslash. That is, instead of a query like "`p(?X) \and q(?X).`" the API requires "`p(?X) \\and q(?X).`".

1.1 The Low-level Interface

The low-level API enables Java programs to send arbitrary queries to $\mathcal{E}$RGO and get results. It is assumed that the following environment variables are set:

`JAVA_BIN`: This variable points to the folder containing the javac and java executable programs. This variable is set in the `windowsVariables.bat` and `unixVariables.sh` scripts in the `java` subfolder of the $\mathcal{E}$RGO distribution.

`PROLOGDIR`: This variable points to the folder containing the XSB executable. This variable is set in the `flora_settings.bat` and `flora_settings.sh` scripts in the `java` folder.

`FLORADIR`: This variable must point to the folder containing the $\mathcal{E}$RGO installation. It is set by the `flora_settings.bat` and `flora_settings.sh` files in the `java` subfolder and this is where users should look in order to get the correct values for their systems. Both of the above files are generated automatically by the system installation scripts.

In order to be able to access $\mathcal{E}$RGO, the Java program must first establish a session for a running instance of $\mathcal{E}$RGO. Multiple sessions can be active at the same time. The knowledge bases in the different running instances are completely independent. Sessions are instances of the class `javaAPI.src.FloraSession`. This class provides methods for opening/closing sessions and loading $\mathcal{E}$RGO knowledge bases (which are also used in the high-level interface). In addition, a session provides methods for executing arbitrary $\mathcal{E}$RGO queries. The following is the complete list of the methods that are available in that class.

- `public FloraSession()`

  This method creates a connection to an instance of $\mathcal{E}$RGO.

- `close()`
  This method must be called to terminate a $\mathcal{E}$RGO session. Note that this does not terminate the Java program that initiated the session: to exit the Java program that talks to $\mathcal{E}$RGO, one needs to execute the statement

  ```
  System.exit();
  ```

  Note that just returning from the `main` method is not enough.

- `public Iterator<FloraObject> ExecuteQuery(String command)`

  This method executes the $\mathcal{E}$RGO command given by the parameter `command`. It is used to execute $\mathcal{E}$RGO queries that do not require variable bindings to be returned back to Java or queries that have only a single variable to be returned. Each binding is represented as an instance of the class `javaAPI.src.FloraObject`. The examples below illustrate how to process the results returned by this method.

- `public Iterator<HashMap<String,FloraObject>> ExecuteQuery(String query, Vector vars)`

  This method executes the $\mathcal{E}$RGO query given by the first argument. The Vector `vars` (of strings) specifies the names of all the variables in the query for which bindings need to be returned. These variables are added to the vector using the method `add` before calling `ExecuteQuery`. For instance, `vars.add("?X")`.

  This version of `ExecuteQuery` returns an iterator over all bindings returned by the $\mathcal{E}$RGO query. Each binding is represented by a `HashMap<String,FloraObject>` object which can be used to obtain the value of each variable in the query (using the `get()` method). The value of each variable returned is an instance of `javaAPI.src.FloraObject`.

  See the examples below for how to handle the results returned by this method.

- `void loadFile(String fileName,String moduleName)`

  This method loads the $\mathcal{E}$RGO program, specified by the parameter `fileName` into the $\mathcal{E}$RGO module specified in `moduleName`.

- `void compileFile(String fileName,String moduleName)`

  This method compiles (but does not load) the $\mathcal{E}$RGO program, specified by the parameter `fileName` for the $\mathcal{E}$RGO module specified in `moduleName`.

- `void addFile(String fileName,String moduleName)`

  This method adds the $\mathcal{E}$RGO program, specified by the parameter `fileName` to an existing $\mathcal{E}$RGO module specified in `moduleName`.

- `void compileaddFile(String fileName,String moduleName)`

  This method compiles the $\mathcal{E}$RGO program, specified by the parameter `fileName` for addition to the $\mathcal{E}$RGO module specified in `moduleName`.

The code snippet below illustrates the low-level API.

Step 1: Writing a $\mathcal{E}$RGO program. Let us assume that we have a file, called `flogic_basics.flr`, which contains the following information:

```
person :: object.
dangerous_hobby :: object.
john:employee.
employee::person.

bob:person.
tim:person.
betty:employee.

person[|age=>integer,
       kids=>person,
       salary(year)=>value,
       hobbies=>hobby,
       believes_in=>something,
       instances => person
|].

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

tim[hobbies -> {stamps, snowboard}].
betty[hobbies->{fishing,diving}].

snowboard:dangerous_hobby.
diving:dangerous_hobby.

?_X[self-> ?_X].

person[|believes_in -> {something, something_else}|].
```


Step 2: Writing a JAVA application to interface with $\mathcal{E}$RGO. The following code loads a $\mathcal{E}$RGO program from a file and then passes queries to the knowledge base.

```
import java.util.*;
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class flogicbasicsExample {

    public static void main(String[] args) {
        // create a new session for a running instance of the engine
        FloraSession session = new FloraSession();
        System.out.println("Engine session started");

        // Assume that Java was called with -DINPUT_FILE=the-file-name
        String fileName = System.getProperty("INPUT_FILE");
        if(fileName == null || fileName.trim().length() == 0) {
            System.out.println("Invalid path to example file!");
            System.exit(0);

        }
        // load the program into module basic_mod
        session.loadFile(fileName,"basic_mod");

        /* Running queries from flogic_basics.flr */

        /* Query for persons */
        String command = "?X:person@basic_mod.";
        System.out.println("Query:"+command);
        Iterator<FloraObject> personObjs = session.ExecuteQuery(command);

        /* Printing out the person names and information about their kids */
        while (personObjs.hasNext()) {
            FloraObject personObj = personObjs.next();
            System.out.println("Person name:"+personObj);


        command = "person[instances -> ?X]@basic_mod.";
        System.out.println("Query:"+command);
        personObjs = session.ExecuteQuery(command);

        /* Prining out the person names  */
        while (personObjs.hasNext()) {
            Object personObj = personObjs.next();
            System.out.println("Person Id: "+personObj);
        }
```



```
        /* Example of ExecuteQuery with two arguments */
        Vector<String> vars = new Vector<String>();
        vars.add("?X");
        vars.add("?Y");

        Iterator<HashMap<String,FloraObject>> allmatches =
            session.ExecuteQuery("?X[believes_in -> ?Y]@basic_mod.",vars);
        System.out.println("Query:?X[believes_in -> ?Y]@basic_mod.");
        while(allmatches.hasNext()) {
            HashMap<String,FloraObject> firstmatch = allmatches.next();
            Object Xobj = firstmatch.get("?X");
            Object Yobj = firstmatch.get("?Y");
            System.out.println(Xobj+" believes in: "+?Yobj);
        }
        // quit the system
        session.close();
        System.exit(0);
    }
}
```

For the information on how to invoke the above Java class in the context of the Java-$\mathcal{E}$RGO API, please see Section 1.3.

1.2 The High-Level Interface

The high-level API operates by creating proxy Java classes for $\mathcal{E}$RGO classes selected by the user. This enables the Java program to operate on $\mathcal{E}$RGO classes by executing appropriate methods on the corresponding proxy Java classes. The use of the high-level API involves a number of steps, as described below.

Readers who intend to use only the low-level Java-$\mathcal{E}$RGO interface can skip this section.

Note: This interface will not work for $\mathcal{E}$RGO programs that use *non-alphanumeric* names for methods and predicates. For instance, if a program involves statements like `foo['bar$#123'->456]` then the interface might generate syntactically incorrect Java proxy classes and errors will be issued during the compilation.

Stage 1: Writing a $\mathcal{E}$RGO file. We assume the same `flogic_basics.flr` file as in the previous example.

Stage 2: Generating Java classes that serve as proxies for $\mathcal{E}$RGO classes. The $\mathcal{E}$RGO side of the Java-to-$\mathcal{E}$RGO high level API provides a predicate to generate Java proxy classes for each F-logic class which have a signature declaration in the $\mathcal{E}$RGO knowledge base. A proxy class gets defined so that it would have methods to manipulate the attributes and methods of the corresponding F-logic class for which signature declarations are available. If an F-logic class has a declared value-returning attribute `foobar` then the proxy class will

have the following methods. Each method name has the form *action*$S_1S_2S_3$_`foobar`, where *action* is either `get`, `set`, or `delete`. The specifier S_1 indicates the type of the method — `V` for value-returning, `B` for Boolean, and `P` for procedural. The specifier S_2 tells whether the operation applies to the signature of the method (`S`), e.g., `person[foobar=>string]`, or to the actual data (`D`), for example, `john[foobar->3]`. Finally, the specifier S_3 tells if the operation applies to the inheritable variant of the method (`I`) or its non-inheritable variant (`N`).

1. `public Iterator<FloraObject> getVDI_foobar()`
   `public Iterator<FloraObject> getVDN_foobar()`
   `public Iterator<FloraObject> getVSI_foobar()`
   `public Iterator<FloraObject> getVSN_foobar()`
   The above methods query the knowledge base and get all answers for the attribute `foobar`. They return iterators through which these answers can be processed one-by-one. Each object returned by the iterator is of type `FloraObject`. The `getVDN` form queries non-inheritable data methods and `getVDI` the inheritable ones. The `getVSI` and `getVSN` forms query the signatures of the attribute `foobar`.

2. `public boolean setVDI_foobar(Vector value)`
   `public boolean setVDN_foobar(Vector value)`
   `public boolean setVSI_foobar(Vector value)`
   `public boolean setVSN_foobar(Vector value)`
   These methods add values to the set of values returned by the attribute `foobar`. The values must be placed in the vector parameter passed these methods. Again, `setVDN` adds data for non-inheritable methods and `setVDI` is used for inheritable methods. `setVSI` and `setVSN` add types to signatures.

3. `public boolean setVDI_foobar(Object value)`
   `public boolean setVDN_foobar(Object value)`
   `public boolean setVSI_foobar(Object value)`
   `public boolean setVSN_foobar(Object value)`
   These methods provide a simplified interface when only one value needs to be added. It works like the earlier set_* methods, except that only one value given as an argument is added.

4. `public boolean deleteVDI_foobar(Vector value)`
   `public boolean deleteVDN_foobar(Vector value)`
   `public boolean deleteVSI_foobar(Vector value)`
   `public boolean deleteVSN_foobar(Vector value)`
   Delete a set of values of the attribute `foobar`. The set is specified in the vector argument.

5. `public boolean deleteVDI_foobar(Object value)`
   `public boolean deleteVDN_foobar(Object value)`
   `public boolean deleteVSI_foobar(Object value)`
   `public boolean deleteVSN_foobar(Object value)`
   A simplified interface for the case when only one value needs to be deleted.

6. `public boolean deleteVDI_foobar()`
   `public boolean deleteVDN_foobar()`

`public boolean deleteVSI_foobar()`
`public boolean deleteVSN_foobar()`
Delete all values for the attribute `foobar`.

For F-logic methods with arguments, the high-level API provides Java methods as above, but they take more arguments to accommodate the parameters that F-logic methods take. Let us assume that the F-logic method is called `foobar2` and it takes parameters `arg1` and `arg2`. As before the `getVDI_*`, `setVDI_*`, etc., forms of the Java methods are for dealing with inheritable $\mathcal{E}$RGO methods and the `getVDN_*`, `setVDN_*`, etc., forms are for dealing with non-inheritable $\mathcal{E}$RGO methods.

1. `public Iterator<FloraObject> getVDI_foobar2(Object arg1, Object arg2)`
   `public Iterator<FloraObject> getVDN_foobar2(Object arg1, Object arg2)`
   Obtain all values for the F-logic method invocation `foobar2(arg1,arg2)`.

2. `public boolean setVDI_foobar2(Object arg1, Object arg2, Vector value)`
   `public boolean setVDN_foobar2(Object arg1, Object arg2, Vector value)`
   Add a set of methods specified in the parameter `value` for the method invocation `foobar2(arg1,arg2)`.

3. `public boolean setVDI_foobar2(Object arg1, Object arg2, Object value)`
   `public boolean setVDN_foobar2(Object arg1, Object arg2, Object value)`
   A simplified interface when only one value is to be added.

4. `public boolean deleteVDI_foobar2(Object arg1, Object arg2, Vector value)`
   `public boolean deleteVDN_foobar2(Object arg1, Object arg2, Vector value)`
   Delete a set of values from `foobar2(arg1,arg2)`. The set is given by the vector parameter `value`.

5. `public boolean deleteVDI_foobar2(Object arg1, Object arg2, Object value)`
   `public boolean deleteVDN_foobar2(Object arg1, Object arg2, Object value)`
   A simplified interface for deleting a single value.

6. `public boolean deleteVDI_foobar2(Object arg1, Object arg2)`
   `public boolean deleteVDN_foobar2(Object arg1, Object arg2)`
   Delete all values for the method invocation `foobar2(arg1,arg2)`.

For Boolean and procedural methods, the generated methods are similar except that there is only one version for the set and delete methods. In addition, Boolean inheritable methods use the `getBDI_*`, `setBDI_*`, etc., form, while non-inheritable methods use the `getBDN_*`, etc., form. Procedural methods use the `getPDI_*`, `getPDN_*`, etc., forms. For instance,

1. `public boolean getBDI_foobar3()`
   `public boolean getBDN_foobar3()`
   `public boolean getPDI_foobar3()`
   `public boolean getPDN_foobar3()`

2. `public boolean setBDI_foobar3()`
   `public boolean setBDN_foobar3()`


```
public boolean setPDI_foobar3()
public boolean setPDN_foobar3()
```

3. ```
   public boolean deleteBDI_foobar3()
   public boolean deleteBDN_foobar3()
   public boolean deletePDI_foobar3()
   public boolean deletePDN_foobar3()
   ```

In addition, the methods to query the ISA hierarchy are available:

- `public Iterator<FloraObject> getDirectInstances()`
- `public Iterator<FloraObject> getInstances()`
- `public Iterator<FloraObject> getDirectSubClasses()`
- `public Iterator<FloraObject> getSubClasses()`
- `public Iterator<FloraObject> getSuperClasses()`
- `public Iterator<FloraObject> getDirectSuperClasses()`

These methods apply to the java proxy object that corresponds to the F-logic class person.

All these methods are generated automatically by executing the following $\mathcal{E}$RGO query (defined in the `javaAPI` package). All arguments in the query must be bound:

```
// write(?Class,?Module,?ProxyClassFileName).
?- write(foo,example,'myproject/foo.java').
```

The first argument specifies the class for which to generate the methods, the file name tells where to put the Java file for the proxy object, and the model argument tells which $\mathcal{E}$RGO model to load this program to. The result of this execution will be the file `foo.java` which should be included with your java program (the program that is going to interface with $\mathcal{E}$RGO). Note that because of the Java conventions, the file name must have the same name as the class name. It is important to remember, however, that proxy methods will be generated only for those F-logic methods that have been declared using signatures.

Let us now come back to our program `flogic_basics.flr` for which we want to use the high-level API. Suppose we want to query the person class. To generate the proxy declarations for that class, we create the file `person.java` for the module `basic_mod` as follows.

```
?- load{'examples/flogic_basics'>>basic_mod}.
?- load{javaAPI}.
?- write(person,basic_mod,'examples/person.java')@\prolog
```

The `write` method will create the file `person.java` shown below. The methods defined in `person.java` are the class constructors for `person`, the methods to query the ISA hierarchy, and the "get", "set" and "delete" methods for each method and attribute declared in the $\mathcal{E}$RGO class `person`. The parameters for the "get", "set" and "delete" Java methods are the


```

same as for the corresponding $\mathcal{E}$RGO methods. The first constructor for class `person` takes a low-level object of class `javaAPI.src.FloraObject` as a parameter. The second parameter is the $\mathcal{E}$RGO module for which the proxy object is to be created. The second `person`-constructor takes F-logic object Id instead of a low-level `FloraObject`. It also takes the module name, as before, but, in addition, it takes a session for a running $\mathcal{E}$RGO instance. The session parameter was not needed for the first `person`-constructor because `FloraObject` is already attached to a concrete session.

It can be seen from the form of the proxy object constructors that proxy objects are attached to specific $\mathcal{E}$RGO modules, which may seem to go against the general philosophy that F-logic objects do not belong to any module — only their methods do. On closer examination, however, attaching high-level proxy Java objects to modules makes perfect sense. Indeed, a proxy object encapsulates operations for manipulating F-logic attributes and methods, which belong to concrete $\mathcal{E}$RGO modules, so the proxy object needs to know which module it operates upon.

**person.java file**

```
import java.util.*;
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class person {

    public FloraObject sourceFloraObject;

    // proxy objects' constructors
    public person(FloraObject sourceFloraObject, String moduleName) { ... }
    public person(String floraOID,String moduleName, FloraSession session) { ... }

    // ISA hierarchy queries
    public Iterator<FloraObject> getDirectInstances() { ... }
    public Iterator<FloraObject> getInstances() { ... }
    public Iterator<FloraObject> getDirectSubClasses() { ... }
    public Iterator<FloraObject> getSubClasses() { ... }
    public Iterator<FloraObject> getDirectSuperClasses() { ... }
    public Iterator<FloraObject> getSuperClasses() { ... }

    // Java methods for manipulating methods
    public boolean setVDI_age(Object value) { ... }
    public boolean setVDN_age(Object value) { ... }
    public Iterator<FloraObject> getVDI_age(){ ... }
    public Iterator<FloraObject> getVDN_age(){ ... }
    public boolean deleteVDI_age(Object value) { ... }
    public boolean deleteVDN_age(Object value) { ... }
    public boolean deleteVDI_age() { ... }
    public boolean deleteVDN_age() { ... }
    public boolean setVDI_salary(Object year,Object value) { ... }
```


```
    public boolean setVDN_salary(Object year,Object value) { ... }
    public Iterator<FloraObject> getVDI_salary(Object year) { ... }
    public Iterator<FloraObject> getVDN_salary(Object year) { ... }
    public boolean deleteVDI_salary(Object year,Object value) { ... }
    public boolean deleteVDN_salary(Object year,Object value) { ... }
    public boolean deleteVDI_salary(Object year) { ... }
    public boolean deleteVDN_salary(Object year) { ... }
    public boolean setVDI_hobbies(Vector value) { ... }
    public boolean setVDN_hobbies(Vector value) { ... }
    public Iterator<FloraObject> getVDI_hobbies(){ ... }
    public Iterator<FloraObject> getVDN_hobbies(){ ... }
    public boolean deleteVDI_hobbies(Vector value) { ... }
    public boolean deleteVDN_hobbies(Vector value) { ... }
    public boolean deleteVDI_hobbies(){ ... }
    public boolean deleteVDN_hobbies(){ ... }
    public boolean setVDI_instances(Vector value) { ... }
    public boolean setVDN_instances(Vector value) { ... }
    public Iterator<FloraObject> getVDI_instances(){ ... }
    public Iterator<FloraObject> getVDN_instances(){ ... }
    public boolean deleteVDI_instances(Vector value) { ... }
    public boolean deleteVDN_instances(Vector value) { ... }
    public boolean deleteVDI_instances(){ ... }
    public boolean deleteVDN_instances(){ ... }
    public boolean setVDI_kids(Vector value) { ... }
    public boolean setVDN_kids(Vector value) { ... }
    public Iterator<FloraObject> getVDI_kids(){ ... }
    public Iterator<FloraObject> getVDN_kids(){ ... }
    public boolean deleteVDI_kids(Vector value) { ... }
    public boolean deleteVDN_kids(Vector value) { ... }
    public boolean deleteVDI_kids(){ ... }
    public boolean deleteVDN_kids(){ ... }
    public boolean setVDI_believes_in(Vector value) { ... }
    public boolean setVDN_believes_in(Vector value) { ... }
    public Iterator<FloraObject> getVDI_believes_in(){ ... }
    public Iterator<FloraObject> getVDN_believes_in(){ ... }
    public boolean deleteVDI_believes_in(Vector value) { ... }
    public boolean deleteVDN_believes_in(Vector value) { ... }
    public boolean deleteVDI_believes_in(){ ... }
    public boolean deleteVDN_believes_in(){ ... }
}
```

Stage 3: Writing Java applications that use the high-level API. The following program (`flogicbasicsExample.java`) shows several queries that use the high-level interface. The class `person.java` is generated at the previous stage. The methods of the high-level interface operate on Java objects that are proxies for $\mathcal{E}$RGO objects. These Java objects are members of the class `javaAPI.src.FloraObject`. Therefore, before one can use the high-level

methods one need to first retrieve the appropriate proxy objects on which to operate. This is done by sending an appropriate query through the method `ExecuteQuery`—the same method that was used in the low-level interface. Alternatively, `person`-objects could be constructed using the 3-argument proxy constructor, which takes F-logic oids.

```
import java.util.*;
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class flogicbasicsExample {

    public static void main(String[] args) {
        /* Initializing the session */
        FloraSession session = new FloraSession();
        System.out.println("Flora session started");

        String fileName = "examples/flogic_basics"; // must be a valid path
        /* Loading the flora file */
        session.loadFile(fileName,"basic_mod");

        // Retrieving instances of the class person through low-level API
        String command = "?X:person@basic_mod.";
        System.out.println("Query:"+command);
        Iterator<FloraObject> personObjs = session.ExecuteQuery(command);

        /* Print out person names and information about their kids */
        person currPerson = null;
        while (personObjs.hasNext()) {
            FloraObject personObj = personObjs.next();
            // Elevate personObj to the higher-level person-object
            currPerson =new person(personObj,"basic_mod");

            /* Set that person's age to 50 */
            currPerson.setVDN_age("50");

            /* Get this person's kids */
            Iterator<FloraObject> kidsItr = currPerson.getVDN_kids();
            while (kidsItr.hasNext()) {
                FloraObject kidObj = kidsItr.next();
                System.out.println("Person: " + personObj + " has kid: " +kidObj);

                person kidPerson = null;
                // Elevate kidObj to kidPerson
                kidPerson = new person(kidObj,"basic_mod");

                /* Get kidPerson's hobbies */
                Iterator<FloraObject> hobbiesItr = kidPerson.getVDN_hobbies();
```



```
                while(hobbiesItr.hasNext()) {
                    FloraObject hobbyObj = hobbiesItr.next();
                    System.out.println("Kid:"+kidObj + " has hobby:" +hobbyObj);
                }
            }
        }

        FloraObject age;
        // create a person-object directly by supplying its F-logic OID
        // father(mary)
        currPerson = new person("father(mary)", "example", session);
        Iterator<FloraObject> maryfatherItr = currPerson.getVDN_age();
        age = maryfatherItr.next();
        System.out.println("Mary's father is " + age + " years old");

        // create a proxy object for the F-logic class person itself
        person personClass = new person("person", "example", session);
        // query its instances through the high-level interface
        Iterator<FloraObject> instanceIter = personClass.getInstances();
        System.out.println("Person instances using high-level API:");
        while (instanceIter.hasNext())
            System.out.println("    " + instanceIter.next());

        session.close();
        System.exit();
    }
}
```

1.3 Executing Java Application Programs with $\mathcal{E}$RGO

To run Java programs that interface with $\mathcal{E}$RGO, follow the following guidelines.

- Place the files `flogicsbasicsExample.java` (the program you have written) and `person.java` (the automatically generated file) in the same directory and compile them using the `javac` command. Add the jar-files containing the API code and `interprolog.jar` to the Java classpath:
  - *FLORADIR*`/java/flora2java.jar`
  - *FLORADIR*`/java/interprolog.jar`

  FLORADIR here should be replaced with the value of the variable `FLORADIR`, which is set by the scripts `flora_settings.sh` (Linux/Mac) or `flora_settings.bat` (Windows), as mentioned in Section 1.1 on page 1.

- Generally, Java programs that call $\mathcal{E}$RGO should be invoked using the following command. For Unix-like systems (Linux, Mac, etc.), change `%`*VAR*`%` to `$`*VAR*:


```
%JAVA_BIN%\java -DPROLOGDIR=%PROLOGDIR%
                -DFLORADIR=%FLORADIR%
                -Djava.library.path=%PROLOGDIR%
                -classpath %MYCLASSPATH% flogicbasicsExample
```

The above command uses several shell variables, which are explained below. Instead of using the variables, one can substitute their values directly.

`JAVA_BIN`: This variable should point to the directory containing the `java` and `javac` executables of the JDK.

`PROLOGDIR`: This variable should be set to the directory containing the XSB executable.

`FLORADIR`: This variable should be set to the directory containing the $\mathcal{E}$RGO system.

`MYCLASSPATH`: This variable should include the jar files containing the API code, i.e., `.../java/flora2java.jar` and file `.../java/interprolog.jar`, plus the above `flogicbasicsExample` class. For instance, it can be set to `%CLASSPATH%;`*FLORADIR*`/java/flora2java.jar;`*FLORADIR*`/java/interprolog.jar;flogicbasicsExample`. For Linux and Mac, use ':' instead of ';' as a separator. As before, *FLORADIR* should be replaced with a proper value, as explained above.

- Some Java applications may employ additional shell variables. For instance, the program that uses the low-level API in Section 1.1 (in Step 2) has the line

  ```
  String fileName = System.getProperty("INPUT_FILE");
  ```

  which means that it expects the shell variable `INPUT_FILE` to be set. In this particular case, it expects that variable to have the address of the `flogic_basics.flr` $\mathcal{E}$RGO file, which it then loads. Therefore, the java command shown above would also need this parameter:

  ```
  -DINPUT_FILE=%INPUT_FILE%
  ```

  In general, one such additional parameter is needed for each property that the Java application queries using the `getProperty()` method.

1.4 Summary of the Variables Used by the Interface

The Java-$\mathcal{E}$RGO interface needs the following shell variables to be set:

- `JAVA_HOME` - this is normally set when you install Java. If not, set this variables manually.
- The following variables can be set by executing the scripts `flora_settings.bat` (Windows) or `flora_settings.sh` (Linux/Mac) located in `flora2/java/`:
  - `FLORADIR` — the path to the $\mathcal{E}$RGO installation directory.
  - `PROLOGDIR` — the path to the folder containing XSB executable.

If you need to set the above variables in some other way, look inside the above scripts to get the exact values these variables should be set to.

- The following variable is set by the scripts `unixVariables.sh` or `windowsVariables.bat`:
  - `JAVA_BIN` — the directory where Java executables are (java, javac).

  If you need to set this variable without running the aforesaid script, you need to know the correct value for that variable. The simplest way is to execute the script and then check the value of environment variable `JAVA_BIN`.

1.5 Building the Prepackaged Examples

Sample applications of the Java-$\mathcal{E}$RGO interface are found in the `java/API/examples` folder. To build the code for the interface, use the scripts `build.bat` or `build.sh` (or `build.bat` on Windows) in the `java/API` folder. To build the the examples, use the scripts `buildExample.sh` or `buildExample.bat` in the `java/API/examples` folder, whichever applies. For instance, to build the `flogicbasicsExample` example, use these commands on Linux, Mac, and other Unix-like systems:

```
cd examples
buildExample.sh flogicbasicsExample
```

On Windows, use this:

```
cd examples
buildExample.bat flogicbasicsExample
```

To run the demos, use the scripts `runExample.sh` or `runExample.bat` in the `java/API/examples` folder. For instance, to run the `flogicbasicsExample`, use this command on Linux, Mac, and the like:

```
runExample.sh flogicbasicsExample
```

On Windows, use this:

```
runExample.bat flogicbasicsExample
```


Chapter 2

$\mathcal{E}$RGO-to-Java Interface: Calling Java from $\mathcal{E}$RGO

by Michael Kifer

This chapter describes the API for opening some standard Java widgets from within $\mathcal{E}$RGO rules. This API also allows one to call arbitrary Java programs and thereby use $\mathcal{E}$RGO for scripting Java applications.

The $\mathcal{E}$RGO-to-Java API works both when $\mathcal{E}$RGO runs as a standalone application and when it is under the control of Ergo Studio. The API calls should work the same in either environment.

2.1 General

The $\mathcal{E}$RGO-to-Java API is available in the system module `\e2j` and calling anything in this module will load that module. If, however, for some reason it is necessary to load this module without executing any operations, one can accomplish this by calling

- `ensure_loaded@\e2j`.

The following additional general API calls are available:

- `System[mode->?Mode] - ?Mode` will be bound to one of the following:
  - `studio` – if $\mathcal{E}$RGO runs as part of Ergo Studio.
  - `[ergo2java,gui]` – if $\mathcal{E}$RGO runs as a standalone mode in an environment that supports graphics. This is usually the case when one invokes $\mathcal{E}$RGO in a command window on a personal computer.
  - `[ergo2java,nogui]` – this is usually the case when $\mathcal{E}$RGO runs in a non-graphical environment, such as a dumb terminal or a command window opened on a remote server. In a `nogui` situation, none of the widgets (windows, dialogs, etc.) will be available. However, the dialog boxes will be simulated through a command-line interface.

- `System[restart]` – restarts the Java subprocess, if it was killed and is needed again. This is required very rarely: for instance, when the Java subprocess was killed outside of $\mathcal{E}$RGO (e.g., via the Task Manager or System Monitor). Java is also killed when `\end` is executed at the $\mathcal{E}$RGO prompt.

- `System[path(studioLogFile)->?File]` – also a rarely used feature. The variable `?File` gets bound to the location of the Studio log file. This calls fails outside of the studio environment. In the future, this API call will be extended to include other file locations that might be deemed useful in the future.

2.2 Dialog Boxes

This part of the API allows the user to pop up various dialog boxes and the find out which button was clicked by the user. Several types of dialog boxes are supported:

- `Dialog[show(?Question)->?Answer]` – pops up a dialog box that asks the user a question and provides an input text field plus the buttons `OK` and `Cancel`. If the user clicks `Cancel` the call fails. Otherwise, if `OK` is clicked, `?Answer` gets bound to whatever the user typed in the input field.

- `Dialog[showOptions(?Title,?Message,?Buttons)->?ChosenButton]` – opens up a dialog box where the user is presented with a number of buttons to click on. Here `?Title` must be bound to an atom—it will be the title of the window; `>Message` is an atom that contains the message to be displayed to the user (e.g., "Please click a suitable button"); and `?Buttons` is a list of labels to appear on the buttons presented as the available choices (e.g., `[Milk,Bread,Honey]`).

- `Dialog[show(?Title,?Message)]` – pops up a dialog box that shows a message (`?Message`) and waits until the user clicks `OK`. `?Title` is the title of the dialog box.

- `Dialog[chooseFile->?File]` – pops up a file chooser. `?File` gets bound to the file chosen by the user.

- `Dialog[chooseFile(?ExtensionsList)->?File]` – like the above, but also takes a parameter that represents a *list* of file extensions. Only the files with that extensions mentioned in the list are shown to the user in the file chooser.

2.3 Windows

This part of the API supports opening, closing, and other operations on windows.

- `Window[open(?WindTitle,?Tooltip)->?Window]` – pops up a new window with the title `?WindTitle` and the tooltip `?Tooltip`. The tooltip is appears when the mouse rests over the window. The variable `?Window` gets bound to the Id of the newly created window. This Id will need to be passed to other API calls that manipulate windows, so the user must usually store these Ids in some predicates.

- `Window[setSize(?Win,?Columns,?Rows)]` – changes the size of the window so it will have the given number of columns and rows. The system will then try to adjust the window (whose Id is passed in the first argument `?Win`) to approximate the requested size.
- `Window[close(?Window)]` – closes the specified window.
- `Window[alive(?Window)]` – tells if the window is alive (i.e., not closed by the user—either programmatically or by clicking the `x` button in the corner of the window).

2.4 Printing to a Window

The following describes how to print to a previously open window and how to erase the window contents.

- `Window[clear(?Window)]` – erases the contents of the given window.
- `Window[print(?Window,?Text)]` – prints `?Text` to a given window. `?Text` specifies what to print and how. Several colors are supported (`black`, `red`, `brown`, `green`, `purple`, `blue`, `magenta`, `orange`, and `default`), as well as a few faces (`italic`, `bold`, `boldital`).

  `?Text` is either a *text descriptor* or a *list* of text descriptors, where a text descriptor is

  – a Hilog term; or
  – *modifier*(Hilog term)

  Here *modifier* is one of the aforesaid colors or faces. Not all faces may be available for the default fonts on your system so, say, `boldital` may appear as `italic` ot as `bold`. Likewise, colors may look different on different screens.

  Note that if you want to print a term like `red(tomato)` then you would have to wrap it in one of the above modifiers, like `default(red(tomato))` (to print `red(tomato)` in the default color—usually black) or `green(red(tomato))` (to print `red(tomato)`). Otherwise, if `red(tomato)` is not wrapped as described, `tomato` will be printed instead.

  Examples. Let us assume that window with Id 3 is open. Then:
  `Window[print(3,magenta('this is red(herring), 1lb'))]@\e2j` will print this is red(herring), 1lb.
  `Window[print(3,[magenta('this is a '), green(2), italic(' pound '), red(herring)])]@\e2j` will print: this is a 2 *pound* herring.

2.5 Scripting Java Applications

The java scripting API allows the user to load Java jar-files, invoke methods that exist in the public classes of those jar-files, and process the results.

- `System[addJar(?Jar)]` – load the specified jar-file into the system.

More details will appear in a later version of this document.

Chapter 3

Querying SQL Databases

by Michael Kifer

This chapter describes the API for SQL queries against relational databases.

3.1 Connecting to a Database

The $\mathcal{E}$RGO-to-SQL API is available in the system module `\sql` and calling anything `@\sql` will load that module. If, for some reason, it is necessary to load this module without executing any operations, one can accomplish this by calling

- `ensure_loaded@\sql`.

Prior to performing any operation on an SQL database the user must *open* a *connection* to that database. $\mathcal{E}$RGO supports two database drivers:

- `odbc`: the general driver to all relational databases that support the ODBC protocol. All major database products and open-source databases support this protocol.[1] The user must be familiar with the basics of setting up ODBC data sources (called DSNs), which specify database drivers and the target databases.
- `mysql`: the native driver for MySQL databases (for Linux, Mac, Windows (64 bit)).

The commands to connect to a database for these two drivers are slightly different.

- The ODBC driver:[2]
  `odbc[open(?ConnectId,?DSN,?User,?Password)]@\sql`.
  Here `?ConnectId` must be bound to a Prolog atom (note: an atom, not a variable) that will henceforth identify the connection. `?DSN` must be bound to an ODBC DSN (data source name), and `?User` and `?Password` must be the user name and the password to

[1] There have been serious problems with ODBC support on Linux and Mac for MySQL server 5.7.

[2] The ODBC driver for MySQL 5.7 has a number of problems on Linux and Mac, so we recommend to use MySQL 5.6, if ODBC is required.

be used to log into the database—both must be Prolog atoms.
Example: `odbc[open(id1,mydbn,me,mypwd)]@\sql.`

- The MySQL driver (Linux, Mac, Windows (64 bit)):
`mysql[open(?ConnectId,?Server,?Database,?User,?Password)]@\sql`.
`?Server` must be bound to the address of the desired database server. Usually this is an IP address such as 123.45.67.89 (with optional port number, e.g., 123.45.67.89:6666) or a domain name, like `abc.example.com` — again with optional port number. On a local machine, the server would usually be just `localhost`.

  The meaning of the other parameters is the same as for the ODBC driver.

  Example: `mysql[open(id2,localhost,test,me,mypwd)]@\sql.`

  Note that one can use the two drivers simultaneously for different connections. However, the connection Ids must be distinct whether the same or different drivers are used. A connection Id can be *reused* if it was previously *closed* (see below).

When done with the database, it is recommended to close the connection to that database for two reasons:

- To avoid hitting the limit of 200 on the number of databases that one can work with at the same time.
- To release the resources allocated by the OS to work with that open connection.

The syntax for closing connections is

```
?ConnectId[close]@\sql.
```

For example, `id2[close]@\sql.`

3.2 Queries

The $\mathcal{E}$RGO-to-SQL API provides a simple query interface to send SQL queries (`SELECT`), updates (`INSERT`, `DELETE`, etc.), schema definition (`CREATE`), and other commands.

- `?ConnectId[query(?QueryId,?QueryList,?ReturnList)]@\sql.`
`?ConnectId` is the Id of a previously open (and not closed) connection. `?QueryId` must be bound to an atom that will represent the query statement that will be created as a result of this command. `?QueryList` is a list that must concatenate into a Prolog atom that forms a valid SQL statement. Components of the list can be variables and terms, and in this way the query can be constructed at run time. `?ReturnList` is a list of variables that must correspond to the list of items in the `SELECT` query. For other types of SQL statements, `?ReturnList` should be an empty list.

  Examples: Assume that our database has a table `Person(name char(40),addr char(100),age integer)`. Then the following is a legal query:


```
?- ?Tbl=Person, ?Age = 33,
   id1[query(qid,['SELECT name, addr FROM ',?Tbl, ' WHERE age=', ?Age],
                 [?Name,?Address]
            )
      ]@\sql.
```

Observe how the SQL query here is constructed at runtime: the table and the value of `age` are bound only when the above $\mathcal{E}$RGO query is executed.

Here is an example of an update statement:

```
id2[query(qa,
          ['insert into Person(name,addr,age)
                        values("mike","unknown",NULL)'],
          []
         )
   ]@\sql.
```

- Preparing queries.
  Frequent databases queries can be precompiled and optimized once and then executed multiple times, which is the recommended modus of operandi. (The previously described query interface is more flexible, but less efficient; it is typically used for infrequent queries or queries that must be constructed at run time, as in the above example.)

  For frequent queries that are known in advance, a two-step process is used. First, the query is *prepared* (i.e., compiled and optimized) and then *executed*. The preparation and execution of such queries allows certain level of flexibility by letting the user to place question marks `?` in lieu of some of the constants (these cannot be column names, table names, variable names, etc. — only regular constants). These question marks can be replaced by actual constants at the query execution time.

  - `?ConnectId[prepare(?QueryId,?QueryList)]@\sql.`
    The meaning of the parameters is the same as before.

    Example:

    ```
    id1[prepare(qid,['SELECT T.addr FROM ', Person,
                     ' T where T.name = ? and T.age = ?']
                )
       ]@\sql.
    ```

    The query Id `qid` can then be used to execute the above query, as shown below.

  - `?QueryId[execute(?BindList,?ReturnList)]@\sql.`
    `?QueryId` must be bound to the query Id of a previously prepared query. `?BindList` must be a list of values that is supposed to be substituted for the `?`'s in the `prepare` command; the `?`'s are substituted in the order in which they appear in the `prepare` statement.

    Example:

    ```
    qid[execute([mike,44],[?Address])]@\sql.
    ```


- Closing query Ids.
  Like database connections, query Ids must be closed in order to release the resources that the OS allocates to the query. There is also a limit of 2000 on the number of active queries, which can be easily reached in applications that query the database heavily. The command for closing the query Ids is:

  ```
  ?QueryId[qclose]@\sql.
  ```

  For instance,

  ```
  qid[qclose]@\sql.
  ```

Finally, we need to mention that when a NULL value is returned as a result of a query, it is returned as a Prolog term `NULL()@\plg`. This implies that if such a term is used as an argument to a literal that is to be inserted into the database, it will be converted to the NULL value.

Chapter 4

Querying SPARQL Endpoints

by Paul Fodor and Michael Kifer

This chapter describes the $\mathcal{E}$RGO interface to SPARQL endpoints (i.e., remote processors that support the SPARQL protocol—both querying and update statements), which is based on Apache Jena. It should be noted from the outset that several triple stores implement SPARQL extensions that go well beyond the SPARQL 1.1 protocol and Jena might not support some of them. The user will see syntax errors whenever such extensions are used in SPARQL queries or update statements.

4.1 General

The $\mathcal{E}$RGO-to-SPARQL API is available through the $\mathcal{E}$RGO system module `\sparql` and calling anything `@\sparql` will load that module. If, however, for some reason it is necessary to load this module without executing any operations, one can accomplish this by calling

- `ensure_loaded@\sparql`.

Prior to performing any queries against a SPARQL endpoint the user must *open* a *connection* to that endpoint. A connection is identified via $\mathcal{E}$RGO symbols, like `MyConnection123`, which are chosen by the user. An endpoint is usually capable of supporting either queries (*query endpoint*) or updates (*update endpoint*), but not both.

- `System[open(?ConnectionId,?EndpointURL,?Username,?Password)]@\sparql`.
  Binds `?ConnectionId` to a *query* endpoint specified by the `?EndpointURL` URL. (See about *update* endpoints below.) `?ConnectionId` must be bound to an $\mathcal{E}$RGO symbol (Prolog atom); it is a connection identifier, and it is chosen by the user. After opening, the connection Id can be used to query the endpoint without re-authentication. `?EndpointURL` must be the URL of a valid *query* endpoint to which the user wishes to connect. It must be an atom. `Username`, and `?Password` must be bound to Prolog atoms ($\mathcal{E}$RGO symbols).
  Example:

`System[open(DBPEDIAConnectionID,'http://dbpedia.org/sparql','','')]@\sparql`.
Binds the symbol `DBPEDIAConnectionID` to the given *query* endpoint with empty credentials (no user id or password). If the connection fails due to an error at the endpoint URL or the user credentials, an error will be issued. If the connection is successful, the query will succeed and one can use `DBPEDIAConnectionID` to query the specified endpoint.

- `System[open(update(MyConnection),'http://localhost:7200/repositories/test/statements','','')]@\sparql`.
  Due to the peculiarities of the SPARQL 1.1 protocol, triple stores usually maintain *different* endpoints (with different URLs!) for query and update operations. So, to both query and update the same triple store one must open two connections. The above form of the `open` statement is used if one wants to connect to an *update* endpoint.

- `System[connectionType(?ConnectionId) -> ?Type]@\sparql`.
  Sometimes one might need to test programmatically if a particular connection is already open and get its connection type. This can be accomplished with the above call.
  If the connection is open, `?Type` gets bound to `query` or `update`—whichever applies. If the connection is not open, the call fails.

- `System[connectionURL(?ConnectionId) -> ?URL]@\sparql`.
  Like `connectionType` but returns the URL of the connection's target endpoint instead of the connection's type.

- `System[close(?ConnectionId)]@\sparql`. `ConnectionId` must be an id of a previously open (and not yet closed) connection to a SPARQL end point. The method closes the connection and releases the space it holds.
  Example:
  `System[close(DBPEDIAConnectionID)]@\sparql`.

It should be noted that closing a connection is usually *not* necessary because each connection involves a relatively small memory overhead and the memory is released when $\mathcal{E}$RGO exits. This only becomes a problem if the user opens (and keeps open) hundreds of thousands connections. The only real inconvenience with keeping many connections open is that one must keep all the names distinct.

Finally, it should be kept in mind that all the definitions and examples in this chapter show $\mathcal{E}$RGO statements in the context of a query or of a rule body. It should be clear that these statements cannot be put in rule heads. If one wants to execute them from within a file, they have to be prefixed with a `?-`, as usual. For instance,

```
?- System[close(DBPEDIAConnectionID)]@\sparql.
```

4.2 Queries and Updates

The $\mathcal{E}$RGO-to-SPARQL API supports several kinds of queries: `select`, `selectAll`, `construct`, `ask`, `describe`, `describeAll`, and `update`. Recall that SPARQL normally uses different endpoints for queries and updates. Accordingly, the first six statements utilize

connections that were previously open and bound to SPARQL *query* endpoints. The last (`update`) statement utilizes connections that are bound to *update* endpoints.

- `Query[select(?ConnectionId,?Query)->?Result]@\sparql`
  runs a SPARQL SELECT `?Query` and successively binds `?Result` to each answer via backtracking. The `?Query` must be an $\mathcal{E}$RGO atom *or* a list. In the former case, the atom must form a valid SPARQL query. In the latter case, the list elements (which typically are $\mathcal{E}$RGO atoms and variables) are converted into atoms and concatenated to form a valid SPARQL query. If the query is not valid, a syntax error is issued. Forming a query using lists is usually necessary only if one wants to pass values through variables from $\mathcal{E}$RGO to the query. The first example below does not pass any variables to the query, so we represent the query simply as an atom. The second example is more interesting, as it passes the $\mathcal{E}$RGO variable `?Subj` into the query and so we use a list.
  Example:
  ```
  Query[select(DBPEDIAConnectionID,'SELECT * WHERE {?x ?r ?y} LIMIT 2')
            -> ?Result]@\sparql.
  ```
  Output:

  ```
  ?Result=["http://www.openlinksw.com/virtrdf-data-formats#default-iid"^^\iri,
           rdf#type,
           "http://www.openlinksw.com/schemas/virtrdf#QuadMapFormat"^^\iri]
  ?Result=["http://www.openlinksw.com/virtrdf-data-formats#default-iid-nullable"^^\iri,
           rdf#type,
           "http://www.openlinksw.com/schemas/virtrdf#QuadMapFormat"^^\iri]
  ```

  Example:
  ```
  ?Subj="http://dbpedia.org/ontology/person"^^\iri,
  Query[select(DBPEDIAConnectionID,
               ['SELECT * WHERE {', ?Subj, '?r ?y} LIMIT 2'])
        -> ?Result]@\sparql.
  ```
  Note that this query passes the binding from the variable `?Subj` into the query. It is important to not confuse $\mathcal{E}$RGO variables, like `?Subj`, with SPARQL variables, like `?r` and `?y`, in the above query. From the $\mathcal{E}$RGO perspective, `?Subj` is a real logical variable and its binding is substituted into the list that forms the query. Without knowing anything about the actual SPARQL variables, $\mathcal{E}$RGO nevertheless "magically" successively binds the variable `?Result` to the lists of pairs $[r_1, y_1], [r_2, y_2], ..., [r_k, y_k]$, where each r_i, y_i are the answers returned by SPARQL. In contrast, `?r` and `?y` are seen by $\mathcal{E}$RGO simply as sequences of characters that form the string `'?r ?y} LIMIT 2'` that becomes part of the query after the list is concatenated. In fact, $\mathcal{E}$RGO does not even look inside that string. From SPARQL perspective, on the other hand, `?r` and `?y` are real variables through which it passes the answers to the query. In contrast, SPARQL does not see the $\mathcal{E}$RGO variable `?Subj` at all, as the binding for that variable becomes part of the query list before the actual query is formed and sent to SPARQL processor.

- `Query[selectAll(?ConnectionId,?Query)->?ResultList]@\sparql`
  runs a SPARQL query, similarly to `select`, except that *all* results are returned at once

in the list `?ResultList`. In contrast, the *select* query returns the results from the query one-by-one. Since we do not pass any values from $\mathcal{E}$RGO to the query, we represent the query simply as an atom.
Example:

```
Query[selectAll(DBPEDIAConnectionID,'SELECT * WHERE {?x ?r ?y} LIMIT 2')
       -> ?ResultList]@\sparql.
```

Output:

```
?Result=[["http://www.openlinksw.com/virtrdf-data-formats#default-iid"^^\iri,
          rdf#type,
          "http://www.openlinksw.com/schemas/virtrdf#QuadMapFormat"^^\iri],
         ["http://www.openlinksw.com/virtrdf-data-formats#default-iid-nullable"^^\iri,
          rdf#type,
          "http://www.openlinksw.com/schemas/virtrdf#QuadMapFormat"^^\iri]]
```

- `Query[construct(?ConnectionId,?Query)->?Result]@\sparql`
  runs a SPARQL `CONSTRUCT` query. As before, `?Query` must be bound either to an atom (which must be a valid CONSTRUCT query) or to a list, which must concatenate into such a valid query. The latter, again, is used to pass values to the query via variables. The CONSTRUCT query is an alternative query to SELECT, that instead of returning a table of results returns an RDF graph. The resulting RDF graph is created by taking the results of the equivalent SELECT query and filling in the values of variables that occur in the CONSTRUCT clause. The resulting graph (a list of triples) is then bound to `?Result`.
  Example:

  ```
  Query[construct(DBPEDIAConnectionID,'CONSTRUCT <http://example3.org/person>
  ?r ?y WHERE ?x ?r ?y LIMIT 2')->?Res]@\sparql.
  ```

  Note that the query refers to a URL constant `<http://example3.org/person>` using the SPARQL syntax for URLs (angle brackets). This syntax differs from the syntax for URLs in $\mathcal{E}$RGO, which is `"http://example3.org/person"^^\iri`. Note that in the second example for SELECT we passed an IRI to the query using the $\mathcal{E}$RGO syntax. $\mathcal{E}$RGO IRIs are converted to SPARQL URLs automatically. However, in that example, we could as well use an atom that represents the desired URL. For instance, `?Subj = '<http://dbpedia.org/ontology/person>'`.

- `Query[ask(?ConnectionId,?Query)]@\sparql`
  runs a SPARQL ASK query. An ASK query tests whether or not a query pattern has a solution. It does not return any results and simply succeeds or fails.
  Example:

  ```
  Query[ask(DBPEDIAConnectionID,'ASK {?x ?prop "Alice"}')]@\sparql.
  ```

  Output: 'Yes' because DBpedia has a matching triple.

- `Query[describe(?ConnectionId,?Query)->?Result]@\sparql`
  runs a SPARQL DESCRIBE query, which returns descriptions of RDF resources. These descriptions are bound to `?Result`.
  Example:

  ```
  Query[describe(DBPEDIAConnectionID,'DESCRIBE ?y WHERE {?x ?r ?y} LIMIT
  1')->?Result]@\sparql.
  ```


- `Query[update(?ConnectionId,?Query)]@\sparql`
  runs update operations on connection `?ConnectionId`, which must be bound to an *update* endpoint. The operations are *insert*, *delete*, *modify*, *load*, and *clear* (described in the standard: `https://www.w3.org/TR/sparql11-update/`). The update requires an update-enabled RDF triple server (e.g., GraphDB, Jena TDB, Virtuoso Universal Server).
  Examples:

```
Query[update(ServerConnectionID,
             'PREFIX dc:  <http://purl.org/dc/elements/1.1/>
              INSERT DATA { <http://example/john> dc:title "A new book" ;
                            dc:creator "A.N.Other" .  }')]@\sparql.
Query[update(ServerConnectionID,
             'PREFIX dc:  <http://purl.org/dc/elements/1.1/>
              DELETE DATA { <http://example/john> dc:title "A new book" ;
                            dc:creator "A.N.Other" .  }')]@\sparql.
```

Here `ServerConnectionID` must be an endpoint that was previously open on an update endpoint.

In addition, there are `constructAll` and `describeAll` queries, which are related to `construct` and `describe` queries the same way `selectAll` is related to `select`: the variable `?Result` gets bound to a list that contains all answers rather than one answer at a time.

Additional examples of queries to standard endpoints (e.g., DBpedia and Wikidata SPARQL endpoints) are provided in Coherent's Ergo Suite Tutorial, in the section on $\mathcal{E}$RGO connectors, at `https://sites.google.com/a/coherentknowledge.com/ergo-suite-tutorial/home/ergo-connectors`.

4.3 Creating Your Own Triple Store

A number of public SPARQL endpoints, such as DBpedia, exist in order to play with SPARQL queries. However, if one wants to modify triples in the store and create endpoints, a local (or a cloud) installation is needed. In this section, we provide the instructions for two triple stores: GraphDB fro Ontotext and Apache's Jena TDB with Fuseki server.

4.3.1 GraphDB

We found that GraphDB from Ontotext (`http://graphdb.ontotext.com/`) is one of the easiest to install, maintain, and experiment with. This is a commercial triple store, but by registering (`http://info.ontotext.com/graphdb-free-ontotext` one can obtain a free license, which supports all major features of the product for small projects. To install GraphDB, use the installation package appropriate for your system. Below are the instructions for Ubuntu Linux (Mint Linux with Cinnamon, to be precise).

After installing the `graphdb-free-7.1.0.deb` package (provided to you by Ontotext after registering), you will find GraphDB in the Programming category in the Start menu. Choosing GraphDB from the menu will open a console and a Firefox browser with a tab open on the

GraphDB workbench. If you don't have Firefox installed, just head to `localhost:7200` in your favorite browser. The Workbench lets you create new triple stores (in the `Admin` menu), put information into the store, and query it. Since we want to query our triple store using $\mathcal{E}$RGO, skip the query/update form: just use the `Admin` menu to create/administer your store.

Let's suppose we created a triple store called `Test`. In response, GraphDB creates two endpoints: `http://localhost:7200/repositories/Test` — a query endpoint and `http://localhost:7200/repositories/Test/statements` — an update endpoint. By opening an $\mathcal{E}$RGO query connection to the former endpoint and an update connection to the latter you will be able to use $\mathcal{E}$RGO to manage your own triple store!

4.3.2 Jena TDB

Jena TDB from Apache is an open source triple store with full support for the SPARQL 1.1 protocol. To install it, visit `http://jena.apache.org/download/#jena-fuseki` and download the latest Apache Jena Fuzeki. As of this writing, the latest release is `apache-jena-fuseki-2.4.0.zip` (or you can choose a tar.gz file).

Unzip the above file in a desired directory (say, `TDB`), change to the directory `TDB/apache-jena-fuseki-2.4.0/` and type

```
fuseki-server --update --mem /test
```

(`fuseki-server.bat` on Windows). This will create an *in-memory* triple store called `test`. Since it is an in-memory store, any data inserted into it will be deleted when the Fuseki server terminates (kill it by typing Ctrl-C). In addition, Fuseki will create two SPARQL endpoints: a query endpoint at `http://localhost:3030/test/query` and an update endpoint at `http://localhost:3030/test/update`. Use these endpoints to perform operations on this triple store via $\mathcal{E}$RGO.

To create a persistent triple store, you need to create a subdirectory in `TDB/apache-jena-fuseki-2.4.0/`, say `MyTestDB` and then start the Fuseki server like this:

```
fuseki-server --update --loc=MyTestDB /test
```

Note that `MyTestDB` is the name of the directory in which to store the data while `test` is the name of the *service*. So, the SPARQL endpoints for this persistent store would be the same as in the previous example: `http://localhost:3030/test/query` and `http://localhost:3030/test/update`.

You can manage this and other triple stores on this server by heading to the Fuseki workbench site at `localhost:3030` in your favorite browser.

To protect the triple stores with a password, edit the file `TDB/apache-jena-fuseki-2.4.0/run/shiro.ini` and add users under the `[users]` section. For instance,

```
[users]
its_me=its_my_pw
```


Chapter 5

Loading RDF and OWL files

by Paul Fodor and Michael Kifer

This chapter describes the $\mathcal{E}$RGO translator and loader of RDF and OWL files (i.e., the Resource Description Framework (RDF) and the Web Ontology Language (OWL) are families of knowledge representation languages for authoring ontologies), which is based on Apache Jena.

5.1 General

The $\mathcal{E}$RGO-to-OWL API is available through the $\mathcal{E}$RGO system module `\owl` and calling anything `@\owl` will load that module. If, however, for some reason it is necessary to load this module without executing any operations, one can accomplish this by calling

- `ensure_loaded@\owl`.

The main predicate for translating and loading RDF and OWL files in Ergo is `rdf_load`:

```
System[rdf_load(?InputFileName, ?InputLangSyntax,
                ?OutputFormat, ?IriPrefixes, ?RdfStorage)]@\owl.
```

The parameters of this query are explained below. They are all input parameters and therefore must be bound. The result of the translation is stored in a place indicated by the last argument and also depends on the output format, as explained below (see the explanations to `?RdfStorage`) .

`?InputFileName` must be bound to an $\mathcal{E}$RGO symbol (Prolog atom); it is an input file name where the RDF or OWL file resides (this can be absolute or relative path). It is advisable that the user uses forward slash as a delimiter. Backslash also works, but it should be doubled as it needs to be escaped.

`?InputLangSyntax` must be bound to an $\mathcal{E}$RGO symbol (Prolog atom); it is an input file syntax: 'RDF/XML', 'JSON-LD' 'TURTLE', 'TTL' , 'N-TRIPLES', 'NT', 'N3', or 'RDF/JSON' (lowercase versions are also accepted).

If `?InputLangSyntax` is an empty atom `''` then the input syntax is guessed from the file extension.

`?OutputFormat` must be bound to `fastload`, `predicates`, or `frames`; it is a flag to select between the output formats. The preferred format is `fastload`, as it is much more efficient for large files.

`?IriPrefixes` must be bound to an $\mathcal{E}$RGO symbol (Prolog atom) and be a sequence or rows, ending with the newline character, where each row has the form `prefix=URL`:

```
'prefix1=URL
 prefix2=URL2
 ...
 prefixN-URL_N'
```

This parameter can be used to define prefixes for compact URIs (curi's) used inside the input RDF/OWL files. These prefixes will be added to the standard pre-defined prefixes `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#`), and `xsd` (`http://www.w3.org/2001/XMLSchema#`). If any of the standard prefixes `rdf`, `rdfs`, `owl`, or `xsd` are also defined in `?IriPrefixes`, the latter override the default definitions.

`?RdfStorage` must be bound to an $\mathcal{E}$RGO symbol (Prolog atom); it indicates where the RDF translation should stored at run time. The nature of that storage depends on the `?OutputFormat` argument. If `?OutputFormat` is bound to `fastload` then `?RdfStorage` is the name of a special $\mathcal{E}$RGO storage and the information in that storage is accessible via the special primitive `fastload{...}` — see the $\mathcal{E}$RGO Programmer's Manual. The example below illustrates that process.

Example:

```
?- System[rdf_load('wine.owl', 'RDF/XML', fastload,
                   '', rdfStorage123)]@\owl,
   fastquery{rdfStorage123, ?P(?S,?O)}.
```

will retrieve all the triples from the file `wine.owl`. Here `?P` will be bound to the property part of RDF triples, `?S` to subjects, and `?O` to objects.

The `fastload` format is a tad harder to query, but it is orders of magnitude faster than the other two formats: `predicates` and `frames`. Therefore it is typically the preferred way to access OWL and RDF. In case of the other two formats, RDF triples are translated into $\mathcal{E}$RGO predicate-shaped facts of the form `prop(subj,obj)` or frame-shaped facts of the form `subj[prop->obj]`. The `?RdfStorage` parameter in that case is the name of the $\mathcal{E}$RGO *module* into which these facts are loaded. Since it is a regular module, the facts there can be queried in the usual way, e.g., `?p(?s,?o)@rdfStorage123`.

Simplified version of the rdf_load query. In most cases the user does not need to use all the options provided by the `rdf_load` method and the following method would suffice:

```
System[rdf_fastload(?InputFileName, ?RdfStorage)]@\owl
```

This method uses `fastload` as the output format. The input language syntax is guessed from the file extension and no IRI prefixes are expected to be supplied. In other words, a call like


```
System[rdf_fastload('wind.owl', rdfStorage123)]@\owl
```

is equivalent to

```
System[rdf_load('wine.owl','',fastload,'',rdfStorage123)]@\owl
```


Chapter 6

Evidential Probabilistic Reasoning in $\mathcal{E}$RGO

by Theresa Swift

Evidential probability [1] is an approach to reasoning about probabilistic information that may be approximate, incomplete, or even contradictory. Rather than providing a full calculus for probabilistic deduction, evidential probability addresses the question of the probability of whether a given object is a member of a given class. To support this, evidential probability extends $\mathcal{E}$RGO with *statistical statements* of the form

$$\backslash\texttt{pct}(targC, refC, Low, High)$$

where $targC$, $refC$ are $\mathcal{E}$RGO classes, while Low and $High$ are numbers between 0 and 1. Such a statement indicates that any given element of $refC$ is an element of $targC$ with probability between $Lower$ and $Upper$. For instance

```
\pct(stolen,redRacing,0.0084,0.0476).
```

could be used to indicate that the proportion of `redRacing` bicycles that are stolen in a given town is between 0.0084 and 0.476. [1]

In order to determine the probability of whether an individual o is in a class C (when o cannot be proved for certain to be in C) statistical statements are used together with Ergo's class membership (`:`/2) and subclass (`::`/2) statements. Information about the classes to which o certainly belongs is extended with statistical information in the following manner. A *candidate* set $Cand$ is collected by examining each statistical \pct-statement S for which o is known to be an element of the reference class of S and for which C is a subclass of the target class of S. Namely,

$$Cand = \{refC | \backslash\texttt{pct}(targC, refC, Low, High),\ C :: targC,\ o \in refC\}$$

Using this candidate set, a series of rules is used to derive a single interval representing the probability that $o \in targC$.

[1] In [1], a more general model is presented, which addresses the question of whether a given n-tuple of domain elements is in the extension of a formula with n free variables.

As mentioned above, evidential probability is good for modelling situations where probabilistic information may be missing or inconsistent. For instance, consider an individual *Mary* in a given knowledge base. *Mary* might belong to a number of different classes: female, mother-of-2, American, resident-of-Virginia, over-40, college-educated, weekend-painter, and so on. To understand the likelihood that *Mary* would contract a given well-studied disease, d, information for various epidemiological studies could be consulted. Some studies, such as those restricted to male subjects, would not apply to Mary because she is not a member of the reference class *Man*. On the other hand, some of the classes to which *Mary* belongs, such as weekend painter, are also irrelevant to whether she will contact d — this time because there would be no \pct-facts with *weekend-painter* as a reference class (presumably because there would be no studies of the relationship between painting on weekends to the disease in question). Of the studies that do pertain to *Mary*, some might be more relevant than others. For instance, a study of the incidence of d for women over 35 would be more relevant than a study of the general population because *Mary* belongs to the class *over-40*, which is more specific than the class of all persons. At the same time, various studies that pertain to *Mary* may conflict with one another. In general, we can't expect there to be a perfect study that considers all potential risk factors for *Mary*. Also, we can't necessarily expect that information from the relevant studies is entirely consistent, due to differences in experimental methods. Thus, evidential probability combines the relevant information, weighs some information more heavily than other information, and resolves conflicts.

The Principles of Evidential Probability One means of weighing information is the principle of *specificity*: a statement S_1 may override statement S_2 if 1) their associated intervals conflict (one interval is not contained in the other); and 2) the reference class of S_1 is more specific to an object o_1 than that of S_2. A second principle is that of *precision*. Given two intervals (L_1, U_1) and (L_2, U_2) where one interval is retained in the other, only the more precise interval is contained. After repeatedly applying the principle of specificity, then of precision, a final candidate set of intervals, S_{fin} is obtained. The final probability is taken to be the smallest interval containing all intervals in S_{fin}.

Evidential probability is thus not a full probabilistic logic, but a meta-logic for defeasible reasoning about statistical statements once non-probabilistic aspects of a model have been derived. It is thus more specialized and less powerful than other types of probabilistic logics; but it is efficient to compute, and applicable to situations where such logics don't apply, due to contradiction, incompleteness, or other factors. [2]

Demonstration Example: Stolen Bikes

The file `.../Ergo/ergo_demos/evidential_probability/bikes.ergo` provides an example of reasoning about evidential probability, and contains a subclass hierarchy along with a set of statistical statements. To use evidential probability, first load the package into the module `ergo_ep`:

[2] Other prioritizations could also be considered, such as prioritizing more trusted information (say, information from better experiments or studies). This type of priority is described in [1] as *sharpening by richness*, but is not implemented here.


```
ergo> [ evidential_probability >> ergo_ep].
```

then load the example

```
ergo> ['ergo_demos/evidential_probability/bikes'].
```

On Windows, use double-backslashes instead of forward slashes:

```
ergo> ['c:ergo_demos\\evidential_probability\\bikes'].
```

At this stage, queries can be made about evidential probability. The query:

```
ergo> \ep(stolen,redRacingImported,?L,?H)@ergo_ep.
```

should return `?L = 0`,`?U = 0.0454`. We show in detail how these bounds bounds were derived. The first step is to *sharpen by specificity*, i.e., to collect all of the relevant statistical statements that pertain to `redRacingImported`, beginning with the most specific. There are no statistical statements about stolen bicycles in the class `redRacingImported`, but there are statements for its immediate superclasses `redRacing`, `racingImported` and `redImported`, all of which form the current *candidate set*. Next, we check statistical statements for the immediate superclasses of the candidate set, namely `red`, `racing` and `imported`. Consider first the interval associated with `red`:`[0.0084,0.0476]`. This interval is considered to conflict with that of e.g., `redRacing`: `[0,0.0454]` since neither interval is contained in the other. In this case, the interval for `red` is overridden and not considered further. Similar considerations override intervals for `imported` and `bike`. Thus, at the end of sharpening by specificity, the candidate classes and their intervals are:

`redRacing`:`[0,0.0454]`, `racing`:`[0,0.0467]`, `redImported`:`[0,0.0467]`, `racingImported`:`[0,0.0582]`.

The next step is to *sharpen by precision*, which throws out all candidate intervals that are contained in other intervals. This step throws out all intervals except for that of `redRacing`: [0,0.0454].

Chapter 7

Importing Tabular Data (CSV, TSV, etc., Files)

by Michael Kifer

This chapter describes the $\mathcal{E}$RGO API for importing tabular data from *delimiter separated values* files (DSV).

A DSV file consists of rows of values that are separated by a *separator*. This is the standard format for exporting tabular data from spreadsheets and other formats. Usually the separator is either a comma or a tab, but could be another character or a sequence of characters. If a field contains spaces, commas, or some other spacial characters, the field is enclosed in *delimiters*. The default is a double quote, e.g., `"a,b| c"`, but can be changed.

The API currently consists of two calls and might be extended in the future. First, the DSV package (`e2dsv`) must be loaded into an $\mathcal{E}$RGO module, say, `dsv`:

```
?- [e2dsv>>dsv].
```

After that, the following predicates will become available:

- `dsv_load`(*?Infile,?Spec,?Format*): The rows of the DSV file, say `'example.csv'`, will be loaded into the predicate specified by *?Spec*. The form of this specification is described below. *?Format* indicates the format of the input file: `csv`, `tsv`, `psv`, or something else, as described below.
- `dsv_save`(*?Infile,?Spec,?OutFile,?Format*): The rows from the CVS *Infile* are converted into the $\mathcal{E}$RGO format (according to *Spec*, which is the same as in `dsv_load`) and then saved in *OutFile*. *?Format* is the same as in `dsv_load` — see below.

The import specification in the above calls can have several forms:

- *predname/arity*: The rows are imported and the predicate *predname/arity* is populated with them. The *arity* piece must equal the number of columns in the typical row of the DSV file. If the DSV file has longer lines, the extra columns will be ignored and

warnings will be issued. If the file has shorter lines than the arity, the extra arguments in *predname* will be padded with variables. All values are imported as general $\mathcal{E}$RGO constant symbols (Prolog atoms).

- *predname*(*ArgSpec1*, ..., *ArgSpecN*): In this form, the user can indicate how the values in the DSV file should be converted. The previous form of *Spec* was importing everything as Prolog atoms, but if the values are numbers then this is not very satisfactory. The possible values for an *ArgSpec* are:
  - `atom` or `?`: the corresponding value from the DSV file is converted into a Prolog atom.
  - `integer`: the value is converted into an integer. If the value cannot be converted into an integer, an error is issued and the value is converted into an atom. The error does not abort the computation and is intended to alert the user.
  - `float`: the value is converted into a floating point/decimal number. If the cannot be converted into a float, an error is issued and the value is converted into an atom. Again, the error is intended to merely alert the user.

  Note: `p/3` is equivalent to the specification `p(atom,atom,atom)` or `p(?,?,?)`.

- *predname*, where *predname* is an atom. In this case, a unary predicate *predname* is populated from the spreadsheet. The predicate will contain lists of values corresponding to each row. The values are all imported as atoms.

  This option is useful when rows are irregular and have different sizes, so it will avoid truncation or padding of the rows during the input.

The argument *?Format* used in the above calls can be either `csv` — for comma-separated files; `tsv` — for tab-separated files; `psv` — for |-separated files; or it can be a list of options of the form:

- `separator="`*chars*`"^^\charlist`; the default is `","^^\charlist`. This is the separator between the fields.
- `delimiter="`*chars*`"^^\charlist`; the default is `"\""^^\charlist`. This is the field delimiter for the fields that contain special characters like commas, spaces, etc. This option is used only if some fields contain double quotes and so the default delimiter will not work.

To query the predicate that is created as a result of the import, the following must be observed:

- The predicate must be queried using the idiom *predname*(...)@*module*, where *module* is the module into which `e2dsv` was loaded (`dsv` in our earlier example). The number of the arguments must match the specification *Spec*—see above.
- The previous contents of the above predicate will be wiped out once the DSV data is loaded.
- If the predicate is queried from within a file rather than the $\mathcal{E}$RGO shell, it must be declared there as


```
:- prolog{predname/arity}.
```

For instance, if the DSV file (CSV, in the example below) is

```
Name,Age,Parent
Bob,13,Mary
Bill,23
```

and we import it as follows:

```
?- [e2dsv>>dsv].
?- dsv_load('example.csv',p/3,csv)@dsv.
```

then the following facts will be added to `p`:

```
?- p(?X,?Y,?Z)@dsv.
?X = Bill
?Y = '13'
?Z = ?

?X = Bob
?Y = '13'
?Z = Mary

?X = Name
?Y = Age
?Z = Parents
```

A warning will be issued regarding Row 3 because it has only two items, while `p` has three arguments.

```
?- dsv_load('example.csv',q,csv)@dsv. // the spec is just an atom
?- q(?X)@dsv.
?X = [Bill,'13']

?X = [Bob,'13',Mary]

?X = [Name,Age,Parents]
```

No warnings will be issued in this case.

If the specification of the output predicate were

```
?- dsv_load('example.csv',p(?,integer,?),csv)@dsv.
```

then the query `p(?X,?Y,?Z)@dsv` would return the result similar to the first example, but `'13'` would be `13` because the numbers in the second column would be imported as numbers rather than atoms. There will be a warning that `Age` in the first row cannot be converted into a number and also a warning concerning the shorter last line in the DSV file.

Chapter 8

Importing JSON Structures

by Michael Kifer

JSON is a popular notation for representing data. JSON is defined by the ECMA-404 standard, which can be found at `http://www.json.org/`. This chapter describes the $\mathcal{E}$RGO facility for importing JSON structures called *values*; it is based on an open source parser called Parson `https://github.com/kgabis/parson`.

8.1 Introduction

In brief, a JSON structure is a *value* is an *object*, an *array*, a *string*, a *number*, `true`, `false`, or `null`. An array is an expression of the form `[`$value_1$`, ..., `$value_n$`]`; an object has a form { $string_1$: $value_1$, ..., $string_n$: $value_n$ }; strings are enclosed in double quotes and are called the *keys* of the object; numbers have the usual syntax, and `true`, `false`, and `null` are constants as written. Here are examples of relatively simple JSON values:

```
{
  "first": "John",
  "last": "Doe",
  "age": 25
}
```

```
[1, 2, {"one" : 1.1, "two": 2.22}, null]
```

```
123
```

and here is a more complex example where values are nested to the depth of five:

```
{
  "status": "ok",
  "results": [{"recordings": [{"id": "12345"}],
              "score": 0.789,
```

```
              "id": "9876"
             }]
}
```

Although not part of the standard, it is quite common to see JSON structures that contains comments like in C, Java, etc. The multiline comments have the form `/* ... */` and the here-to-end-of-line comments start with the `//`. $\mathcal{E}$RGO ignores such comments.

The standard recommends, but does not require that the keys in an object do not have duplicates (at the same level of nesting). Thus, for instance,

```
{"a":1, "b":2, "b":3}
```

is allowed, but discouraged. By default, the $\mathcal{E}$RGO parser does not allow duplicate keys and considers such objects as ill-formed. However, it also provides a way to set an option to allow duplicate keys.

8.2 API for Importing JSON as Terms

When $\mathcal{E}$RGO ingests a JSON structure, it represents it as a term as follows:

- Arrays are represented as lists.
- Strings are represented as $\mathcal{E}$RGO symbols (Prolog atoms).
- Numbers are represented as such.
- `true`, `false`, `null` are represented as the Prolog (not HiLog!) 0-ary terms of the form `true()`, `false()`, and `'NULL'()`.
- Finally, an object of the form $\{ str_1 : val_1, \ldots, str_n : val_n \}$ is represented as `json([`str'_1`=`val'_1`,...,`str'_n`=`val'_n`])`, where str'_i is the atom corresponding to the string str_i and val'_i is the $\mathcal{E}$RGO representation of the JSON value val_i. Here, as above, `json` is a unary Prolog, not HiLog, function symbol.

For instance, the above examples would be represented as $\mathcal{E}$RGO terms as follows:

```
json([first = John, last = Doe, age = 25])@\prolog
[1, 2, json([one = 1.1000, two = 2.2200])@\prolog, NULL()@\prolog]
123
json([status = ok,
      results = [json([recordings = [json([id = '12345'])@\prolog],
                       score = 0.7890,
                       id = '9876']
                      )@\prolog]
     ])@\prolog
```


where we tried to pretty-print the last result so it would be easier to relate to the original (which was also pretty-printed).

$\mathcal{E}$RGO provides the following methods for importing JSON:

- *Source*`[parse -> ?`*Result*`]@\json`
  Here *Source* can have the form `string(`*Atom*`)`, `str(`*Atom*`)`, `file(`*Atom*`)`, *Atom*, or a variable. The forms `string(`*Atom*`)` and `str(`*Atom*`)` must supply an atom whose content is a JSON structure and *Result* will then be bound to the $\mathcal{E}$RGO representation of that structure. The forms `file(`*Atom*`)` and *Atom* interpret *Atom* as a file name and will read the JSON structure from there. The last form, when the source is a variable, assumes that the JSON structure is provided via the standard input. The user will have to send the end-of-file signal (Ctrl-D in Linux or Mac; Ctrl-Z in Windows) in order to tell the when the entire term has been entered.[1] If the input JSON structure contains a syntax error or some other problem is encountered (e.g., not enough memory) then the above predicate will fail and a warning indicating the reason will be printed to the standard output.

  ?Result can be a variable or any other term. If *?Result* has the form `pretty(`*something*`)` then *something* gets unified with a pretty-printed string representation of the input JSON structure. If *?Result* has any other form (typically a variable) then the input is converted into an $\mathcal{E}$RGO term as explained above. For instance, the query `string('{"abc":1, "cde":2}')[parse->?X]@\json` will bind `?X` to the term `json([abc=1,cde=2])@\prolog` while the query `string('{"abc":1, "cde":2}')[parse->pretty(?X)]@\json` will bind `?X` to the atom

```
'{
    "abc": 1,
    "cde": 2
}'
```

  which is a pretty-printed copy of the input JSON string.

- *Source*`[parse(`*Selector*`) -> ?`*Result*`]@\json`
  The meaning of *Source* and *Result* parameters here are the same as before. The *Selector* parameter must be a path expression of the form "string1.string2.string3" (with one or more components) that allows one to select the *first* sub-object of a bigger JSON object and return its representation. Note, the first argument *must* supply an object, not an array or some other type of value. For instance, if the input is

```
{ "first":1, "second":{"third":[1,2], "fourth":{"fifth":3}} }
```

  then the query `?[parse(first) -> ?X]@\json` will bind `?X` to 1 while `?[parse('second.fourth') -> ?X]@\json` will bind it to `json([fifth = 3])@\prolog`.

  Note that the selector lets one navigate through subobjects and not through arrays. If an array is encountered in the middle, the query will fail. For instance, if the input is

[1] Sending the end-of-file signal is not possible in the $\mathcal{E}$RGO Studio Listener, so this last option is not available through the studio.


```
{ "first":1, "second":[{"third":[1,2], "fourth":{"fifth":3}}] }
```

then the query `?[parse('second.fourth') -> ?X]@\json` will fail and `?X` will not be bound to anything because the selector `"second"` points to an array and the selector `"fourth"` cannot penetrate it.

Also note that if the JSON structure has more than one sub-object that satisfies the selection and duplicate keys are allowed (e.g., in {`"a":1, "a":2`} both 1 and 2 satisfy the selection) then only the first sub-object will be returned. (See below to learn about duplicate keys in JSON.)

- `set_option(`*option=value*`)@\json`
  This sets options for parsing JSON for all the subsequent calls to the `\json` module. Currently, only the following is supported:

  ```
  duplicate_keys=true
  duplicate_keys=false
  ```

  As explained earlier, the default is that duplicate keys in JSON objects are treated as syntax errors. The first of the above options tells the parser to allow the duplicates. The second option restores the default.

Here is a more complex example, which uses the JSON parser to process the result of a search of Google's Knowledge Graph to see what it knows about Benjamin Grosof. To make the output a bit more manageable, we are only asking to get the JSON subobject rooted at the property `itemListElement`. The Knowledge Graph itself is queried using XSB's `curl` library.

```
?- load_page(url('https://kgsearch.googleapis.com/v1/entities:search?query=
benjamin_grosof&key=AIzaSyAaMs1AEkgRGAs_hkcULQLJ5NKrEOzyOBO&limit=1'),
             [secure(false)], ?, ?_SearchResult, ?)@\plgall(curl),
   str(?_SearchResult)[parse(itemListElement) -> ?Answer]@\json.
```

The answer to this query is

```
?Answer = [json(['@type' = EntitySearchResult,
                 result = json(['@id' = 'kg:/m/09pb9y8',
                                name = 'Benjamin Nathan Grosof',
                                '@type' = [Person, Thing],
                                description = Mathematician]),
                 resultScore = 19.3944])]
```

8.3 API for Importing JSON as Facts

The API for importing JSON as terms is useful if one needs to traverse the imported JSON tree structure and process it in some complex way. However, in knowledge interchange, JSON is often used to exchange facts about enterprises being modeled by the different knowledge base. For instance, the native representation in Wikidata and MongoDB is JSON and to get

the Wikidata or the MongoDB facts into $\mathcal{E}$RGO we would want to represent the information as queriable facts. Fortunately, converting JSON into $\mathcal{E}$RGO facts is easy because the former is mappable 1-1 to $\mathcal{E}$RGO frames. For instance, the following JSON

```
{"kind": "person", "fullName": "John Doe", "age": 22, "gender": "Male",
 "child": {{"fullName":"Bob Doe", "age":1},    // embedded JSON objects
           {"fullName":"Alice Doe", "age":3}},
 "citiesLived": [{ "place":"Boston", "numberOfYears":5}, // JSON objects
                 {"place":"Rome", "numberOfYears":6}]}   // embedded in list
```

translates into this:

```
\#[kind->person, fullName->'John Doe', age->22, gender->Male,
   child->{\#[fullName->'Bob Doe', age->1],
           \#[fullName->'Alice Doe', age->3]},
   citiesLived->[\#[place->Boston, numberOfYears->5],
                 \#[place->Rome, numberOfYears->6]]
].
```

The principle should be obvious from the above example except that frames are not allowed inside lists, and so

```
[\#[place->Boston,numberOfYears->5], \#[place->Rome,numberOfYears->6]]
```

is not a valid $\mathcal{E}$RGO syntax. However, this is easy to fix by converting the above list with embedded frames into the following list plus additional frame-facts, where `newObjId1` and `newObjId2` are newly invented constants that do not appear anywhere else:

```
 [newObjId1, newObjId2]  // complex list became simple
newObjId1[place->Boston, numberOfYears->5].  // these facts were
newObjId2[place->Rome, numberOfYears->6].    // embedded in the above list
```

Thus, the actual translation of the JSON structure in question is

```
\#[kind->person, fullName->'John Doe', age->22, gender->Male,
   child->{\#[fullName->'Bob Doe', age->1],
           \#[fullName->'Alice Doe', age->3]},
   citiesLived->[newObjId1, newObjId2]  // list no longer has embedded frames
].
newObjId1[place->Boston, numberOfYears->5].  // frames formerly
newObjId2[place->Rome, numberOfYears->6].    // embedded in a list
```

Conversion of JSON structures into facts is done by the following API calls:

- `?Src[parse2memory(?Mod)]@\json`: The meaning of `?Src` is as before. This API call takes the input JSON structure, which must be a JSON object (and not a list, number, etc.) and inserts facts, as explained above, into the $\mathcal{E}$RGO module `?Mod`, which must exist beforehand (e.g., created via `newmodule{...}`).

- `?Src[parse2memory(?Mod,?Selector)]@\json`: Like the previous call but also takes the selector argument whose meaning is as in the case of the term-based JSON import.
- `?Src[parse2file(?File)@\json]`: This is similar to parsing to memory, but the facts are instead written to the specified file. If the file already exists, it is erased first. The file can then be loaded or added into some $\mathcal{E}$RGO module (adding is recommended).
- `?Src[parse2file(?File,?Selector)@\json]`: Like the previous case, but also takes the selector argument.

8.4 Exporting to JSON

$\mathcal{E}$RGO provides API for exporting HiLog terms as well as objects to JSON.

8.4.1 Exporting HiLog Terms to JSON

The case of terms is simple: a term is represented simply as a JSON object with two features: *functor* and `arguments`. The functor is also a term so it is futher converted according to the same rules. The arguments part is a list of terms and the latter are converted recursively by the same rule. For instance,

```
p(o)(${a(9)},b,?L,[pp(ii),2,3,?L])[term2json -> ?X]@\json.

?X = '{"functor":{"functor":"p","arguments":["o"]}
       "arguments":[{"predicate":"a","module":"main","arguments":[9]},
                    "b",
                    {"variable":"h0"},
                    [{"functor":"pp","arguments":["ii"]},
                    2,
                    3,
                    {"variable":"h0"}]]}'
```

Note that a term can be a reified predicate in which case the `"predicate"` feature name is used instead of `"functor"`. Also, a variable is translated into a JSON object of the form {`"variable": "varname"`}. Since variable names in a logic formula are immaterial and all that matters is whether two variables are the same or not, only internal names are shown. In the above example, the two occurrences of `?X` are shown as `"h0"`.

- `?Term[term2json -> ?Json]@\json` — convert HiLog term to which `?Term` is bound into a JSON expression. The result is an atom (an $\mathcal{E}$RGO symbol) that contains the JSON expression. Such an atom can be sent to a JSON-aware external application.

8.4.2 Exporting $\mathcal{E}$RGO Objects to JSON

This API takes a HiLog term that is interpreted as an object Id and returns the JSON encoding of all the immediate superclasses of that object and all the properties of that object.

The input object can be in the current module or in some other module. Furthermore, the API can take conditions that would filter out the properties of the object that we are looking for as well as eliminate the descendant object that we don't want to see in the JSON encoding. The idea of the encoding can best be understood via examples.

The first example gives a JSON encoding for the object `kati` from the `family_obj.flr` demo located in the `demos/` folder in the $\mathcal{E}$RGO distribution. First, we need to load this demo via the command `demo{family_obj}`. To get the JSON encoding, we use the `object2json` method and then pretty-print the result as explained previously. That is,

```
?- demo{family_obj},
   set_option(duplicate_keys=true)@\json,
   kati[object2json -> ?Json]@\json,
   string(?Json)[parse->pretty(?Res)]@\json,
   writeln(?Res)@\io.

{
        "\\self": "kati",
        "\\isa": [
            "female"
        ],
        "ancestor": "hermann",
        "ancestor": "johanna",
        "ancestor": "rita",
        "ancestor": "wilhelm",
        "brother": "bernhard",
        "brother": "karl",
        "daughter": "eva",
        "father": "hermann",
        "mother": "johanna",
        "parent": "hermann",
        "parent": "johanna",
        "sister_in_law": "christina",
        "uncle": "franz",
        "uncle": "heinz"
}
```

Note that we set the `duplicate_keys=true` option because in the `family_obj` demo most of the properties (like `ancestor`) are multi-valued, which leads to repeated keys in JSON representation. As we noted, this is allowed, but some applications do not support such JSON expressions. If one needs to talk to such applications, simply don't set the `duplicate_keys=true` option and the above will represent duplicate JSON keys using lists. For instance, `"ancestor":["hermann","johanna","rita","wilhelm"]`. Note, however, that without the `duplicate_keys` option the JSON encoding becomes lossy, since we no longer can tell whether the original $\mathcal{E}$RGO attribute `ancestor` was multivalued (with each single value being a string) or it was single-valued and the value was an ordered list.

Here we also note that the use of JSON API can often be simpler if one recalls the very useful

syntax of path expressions. For instance, the 3d and 4th lines in the above query can be written much more shortly as

```
string(kati.object2json)[parse->pretty(?Res)]@\json
```

If we try to encode the class `female` we get the following:

```
string(kati.object2json)[parse->pretty(?Res)]@\json, writeln(?Res)@\io.
{
 "\\self": "female",
 "\\sub": [
     "person"
 ],
 "type": "gender"
}
```

Note that in $\mathcal{E}$RGO properties can be HiLog terms and so they cannot be encoded simply as a string like `"parent"`. For instance,

```
?- insert{{a,b}:{c,d},d::k, k[|eee(123)->kkk|]}.
?- a[object2json -> ?Json]@\json,
   string(?Json)[parse->pretty(?Res)]@\json,
   writeln(?Res)@\io.

{
 "\\self": "a",
 "\\isa": [
     "c",
     "d"
 ],
 "\\keyval": [
     {
         "functor": "eee",
         "arguments": [
             123
         ]
     },
     [
         "kkk"
     ]
 ]
}
```

Note that `eee(123) -> kkk` is a complex property that object `a` inherits from class `k`. It is encoded as a JSON keypair `"\\keyval"` : *list* where the first element of *list* is the encoding of `eee(123)` and the second of `"kkk"`.

Now we are ready to present the different versions of the `object2json` method.

- `?Obj[object2json -> ?Json]@\json` — take an object and return a Prolog atom that contains a JSON representation of the object's immediate superclasses and properties with respect to the $\mathcal{E}$RGO module where this call is made.
- `?Obj[object2json(?Module) -> ?Json]@\json` — as above, but the properties and the superclasses of `?Obj` are taken from the module `?Module`.
- `?Obj[object2json(?Mod)(?keyFilter,?valFilter,?classFilter)->?Json]@\json` — this version lets one to not only specify the module but also impose conditions on the properties of `?Obj`, on the superclasses, and on the property values that we want to see in the JSON representation. In the above, `(?Mod)` can be omitted and the current module will be used then. A `null` (or any other constant) condition means "no filtering for that type of argument." Otherwise, the filters must be unary predicates or primitives. In the example below we use unary primitives `isnumber{?}` and `isatom{?}`.

First, we show what happens without filtering. It is an expansion of an earlier example:

```
ergo> insert{{a,b}:c, c::{h,k}, h[|www->1|],k[|ppp->kk, eee(123)->kkk|]},
      string(a.object2json)[parse->pretty(?Res)]@\json, writeln(?Res)@\io.
{
    "\\self": "a",
    "\\isa": [
        "c"
    ],
    "ppp": [
        "kk"
    ],
    "www": [
        1
    ],
    "\\keyval": [
        {
            "functor": "eee",
            "arguments": [
                123
            ]
        },
        [
            "kkk"
        ]
    ]
}
```

In contrast, the following query says that we want to see only the atomic properties (so `eee(123)` will be omitted) and only such properties whose values are numbers. No restrictions on superclasses is imposed:

```
ergo> string(a.object2json(isatomic{?},isnumber{?},null))[
```



```
                parse->pretty(?Res)
      ]@\json, writeln(?Res)@\io.
{
    "\\self": "a",
    "\\isa": [
        "c"
    ],
    "www": [
        1
    ]
}
```

We see that the complex property `eee(123)->1` got dropped because it is not atomic and the property `"ppp"` got dropped because its values are not integers.

Recursive export. Sometimes it is desirable to convert not just an object, but an object together with its descendant objects—the ones reachable from the object via its attributes—into a single JSON structure. For instance, in our `family_obj.flr` example, `kati` has an `ancestor`-descendant object `hermann`, which is also a person-object that has its own JSON representation. We might want to attach that representation to the `kati`-JSON structure at the point where `"hermann"` is attached. To enable such a *recursive* export into JSON, one must set the `recursive_export` option by executing the following query:

```
?- set_option(recursive_export=true)@\json.
```

We cannot show here the result of a recursive export for `kati`, as the resulting structure is too big, but we will show a smaller example:

```
ergo> insert{{a,b}:d, d::e, e::k ,k[|ppp->kk:d[prop1->abc,prop2->3], ppp->jj|]},
      string(a.object2json)[parse->pretty(?_Res)]@\json, writeln(?_Res)@\io.
 {
     "\\self": "a",
     "\\isa": [
         "d"
     ],
     "ppp": [
         {
             "\\self": "jj"
         },
         {
             "\\self": "kk",
             "\\isa": [
                 "d"
             ],
             "ppp": [
                 {
```



```
                    "\\self": "jj"
                },
                {
                    "\\self": "kk",
                    "\\isa": [
                        "d"
                    ]
                }
            ],
            "prop1": [
                {
                    "\\self": "abc"
                }
            ],
            "prop2": [
                {
                    "\\self": 3
                }
            ]
        }
    ]
}
```

Here we see that `"kk"` (a `ppp`-descendant object of `"a"`) is also JSON-expanded. Moreover, it is easy to see that `kk[ppp->kk]` is true, which means that `kk` is a `ppp`-descendant of itself. Thus, there is a cycle through `kk` in the descendant-object relation and if we kept expanding `kk` as we traverse the `ppp` attribute, the resulting JSON term would be infinite. Therefore, as you can see, the second time we encounter `"kk"` it is *not* expanded and only its isa-information is shown (the sub-information would have also been shown, if it existed).

Chapter 9

Persistent Modules

by Vishal Chowdhary

This chapter describes a $\mathcal{E}$RGO package that enables persistent modules. A *persistent module* (abbr., PM) is like any other $\mathcal{E}$RGO module except that it is associated with a database. Any insertion or deletion of base facts in such a module results in a corresponding operation on the associated database. This data persists across $\mathcal{E}$RGO sessions, so the data that was present in such a module is restored when the system restarts and the module is reloaded.

9.1 PM Interface

A module becomes persistent by executing a statement that associates the module with an ODBC data source described by a DSN. To start using the module persistence feature, first load the following package into some module. For instance:

```
?- [persistentmodules>>pm].
```

The following API is available. Note that if you load `persistentmodules` into some other module, say `foo`, then `foo` should be used instead of `pm` in the examples below.

- `?- ?Module[attach(?DSN,?DB,?User,?Password)]@pm.`
  This action associates the data source described by an ODBC `DSN` with the module. If `?DB` is a variable then the database is taken from the DSN. If `?DB` is bound to an atomic string, then that particular database is used. Not all DBMSs support the operation of replacing the DSN's database at run time. For instance, MS Access or PostgresSQL do not. In this case, `?DB` must stay unbound or else an error will be issued. For other DBMS, such as MySQL, SQL Server, and Oracle, `?DB` can be bound.

  The `?User` and `?Password` must be bound to the user name and the password to be used to connect to the database.

  The database specified by the DSN must already exist and must be created by a previous call to the method `attachNew` described below. Otherwise, the operation is aborted. The database used in the `attach` statement must not be accessed directly—only through

the persistent modules interface. The above statement will create the necessary tables in the database, if they are not already present.

Note that the same database can be associated with several different modules. The package will not mix up the facts that belong to different modules.

- `?- ?Module[attachNew(?DSN,?DB,?User,?Password)]@pm.`
  Like `attach`, but a new database is created as specified by `?DSN`. If the same database already exists, an exception of the form `ERGO_DB_EXCEPTION(?ErrorMsg)` is thrown. (In a program, include `flora_exceptions.flh` to define `ERGO_DB_EXCEPTION`; in the shell, use the symbol `'_$ergo_db_error'`.) This method creates all the necessary tables, if they are not already present.

  Note that this command works only with database systems that understand the SQL command `CREATE DATABASE`. For instance, MS Access does not support this command and will cause an error.

- `?- ?Module[detach]@pm.`
  Detaches the module from its database. The module is no longer persistent in the sense that subsequent changes are not reflected in any database. However, the earlier data is not lost. It stays in the database and the module can be reattached to that database.

- `?- ?Module[loadDB]@pm.`
  On re-associating a module with a database (i.e., when `?Module[attach(?DSN, ?DB,?User,?Password)]@pm` is called in a new $\mathcal{E}$RGO session), database facts previously associated with the module are loaded back into it. However, since the database may be large, $\mathcal{E}$RGO does not preload it into the main memory. Instead, facts are loaded on-demand. If it is desired to have all these facts in main memory at once, the user can execute the above command. If no previous association between the module and a database is found, an exception is thrown.

- `?- set_field_type(?Type)@pm.`
  By default, $\mathcal{E}$RGO creates tables with the VARCHAR field type because this is the only type that is accepted by all major database systems. However, ideally, the CLOB (character large object) type should be used because VARCHAR fields are limited to 4000-7000 characters, which is usually inadequate for most needs. Unfortunately, the different database systems differ in how they support CLOBs, so the above call is provided to let the user specify the field types that would be acceptable to the system(s) at hand. The call should be made right before `attachNew` is used. Examples:

```
?- set_field_type('TEXT DEFAULT NULL')@pm.   // MySQL, PostgreSQL
?- set_field_type('CLOB DEFAULT NULL')@pm.   // Oracle, DB2
```

Once a database is associated with the module, querying and insertion of the data into the module is done as in the case of regular (transient) modules. Therefore PM's provide a transparent and natural access to the database and every query or update may, in principle, involve a database operation. For example, a query like `?- ?D[dept -> ped]@StonyBrook.` may invoke the SQL `SELECT` operation if module `StonyBrook` is associated with a database. Similarly `insert{a[b -> c]@stonyBrook}` and `delete{a[e -> f]@stonyBrook}` will invoke

SQL `INSERT` and `DELETE` commands, respectively. Thus, PM's provide a high-level abstraction over the external database.

Note that if `?Module[loadDB]@pm` has been previously executed, queries to a persistent module will *not* access the database since $\mathcal{E}$RGO will use its in-memory cache instead. However, insertion and deletion of facts in such a module will still cause database operations.

9.2 Examples

Consider the following scenario sequence of operations.

```
// Create new modules mod, db_mod1, db_mod2.
ergo>  newmodulemod, newmoduledb_mod1, newmoduledb_mod2.
ergo>  [persistentmodules>>pm].

// insert data into all three modules.
ergo>  insertq(a)@mod,q(b)@mod,p(a,a)@mod.
ergo>  insertp(a,a)@db_mod1, p(a,b)@db_mod1.
ergo>  insertq(a)@db_mod2,q(b)@db_mod2,q(c)@db_mod2.

//  Associate modules db_mod1, db_mod2 with an existing database db
//  The data source is described by the DSN mydb.
ergo>  db_mod1[attach(mydb,db,user,pwd)]@pm.
ergo>  db_mod2[attach(mydb,db,user,pwd)]@pm.

// insert more data into db_mod2 and mod.
ergo>  inserta(p(a,b,c),d)@db_mod2.
ergo>  insertq(a)@mod,q(b)@mod,p(a,a)@mod.

// shut down the engine
ergo>  \halt.
```

Restart the $\mathcal{E}$RGO system.

```
// Create the same modules again
ergo>  newmodulemod, newmoduledb_mod1, newmoduledb_mod2.

// try to query the data in any of these modules.
ergo>  q(?X)@mod.
No.

ergo>  p(?X,?Y)@db_mod1.
No.

//  Attach the earlier database to db_mod1.
ergo>  [persistentmodules>>pm].
```



```
ergo>  db_mod1[attach(mydb,db,user,pwd)]@pm.

// try querying again...

// Module mod is still not associated with any database and nothing was
// inserted there even transiently, we have:
ergo>  q(?X)@mod.
No.

// But the following query retrieves data from the database associated
// with db_mod1.
ergo>  p(?X,?Y)@db_mod1.
?X = a,
?Y = a.

?X = a,
?Y = b.

Yes.

// Since db_mod2 was not re-attached to its database,
// it still has no data, and the query fails.
ergo>  q(?X)@db_mod2.

No.
```


Chapter 10

SGML and XML Parsers for $\mathcal{E}$RGO

by Rohan Shirwaikar

This chapter documents the $\mathcal{E}$RGO package that provides SGML and XPath parsing capabilities. One set of predicates supports parsing SGML, XML, and HTML documents, which creates $\mathcal{E}$RGO objects in the user specified module. Other predicates evaluate XPath queries on XML documents and create $\mathcal{E}$RGO objects in user specified modules. The predicates make use of the **sgml** and **xpath** packages of XSB.

10.1 Summary of the Predicates

load_xml_structure/3	Parse XML data into $\mathcal{E}$RGO objects
load_sgml_structure/3	Parse SGML data into $\mathcal{E}$RGO objects
load_html_structure/3	Parse HTML data into $\mathcal{E}$RGO objects
load_xhtml_structure/3	Parse XHTML data into $\mathcal{E}$RGO objects
parse_xpath_xml/5	Apply an XPath expression to an XML document and parse the result
parse_xpath_sgml/5	Apply an XPath expression to an SGML document and parse the result
parse_xpath_html/5	Apply an XPath expression to an HTML document and parse the result
parse_xpath_xhtml/5	Apply an XPath expression to an XHTML document and parse the result

10.2 Description

This package supports parsing SGML, XML, and HTML documents, converting them to sets of $\mathcal{E}$RGO objects stored in user-specified $\mathcal{E}$RGO modules. The SGML interface provides facilities to parse input in the form of files, URLs and strings (Prolog atoms).

For example, the following XML snippet

```
<greeting id='1'>
<first ssn=111'>
John
</first>
```

```
</greeting>
```

will be converted into the following $\mathcal{E}$RGO objects:

```
o1[ element -> {o2}]
o2[ id -> '1']
o2[ first -> {o3}]
o2[ ssn -> '111']
```

To load the `flrxml` package, the user should run the following command at the $\mathcal{E}$RGO prompt.

```
ergo>  [flrxml].
```

The following predicates are provided by the `flrxml` package. They take SGML, XML, HTML, or XHTML documents and create the corresponding $\mathcal{E}$RGO objects as specified in Section 10.4.

load_sgml_structure(`+?Source,-?Warn,+?Module`)**@flrxml**

load_xml_structure(`+?Source,-?Warn,+?Module`)**@flrxml**

load_html_structure(`+?Source,-?Warn,+?Module`)**@flrxml**

load_xhtml_structure(`+?Source,-?Warn,+?Module`)**@flrxml**

The arguments to these predicates have the following meaning:

`?Source` is an input SGML, XML, HTML, or XHTML document. It is of the form `url(url)`, `file('file name')` or `string('document as a string')`. `?Module` is the name of the $\mathcal{E}$RGO module where the objects created by `flrxml` should be placed. `?Module` must be bound. `?Warn` gets bound to a list of warnings, if any are generated, or to an empty list.

10.3 XPath Support

XPath support is based on the XSB `xpath` package, which must be configured as explained in the XSB manual. This package, in turn, relies on the XML parser called `libxml2`. It comes with most Linux distributions and is also available for Windows, MacOS, and other Unix-based systems from `http://xmlsoft.org`. Note that both the library itself and the `.h` files of that library must be installed.

The following predicates are provided. They select parts of the input document using the provided XPath expression and create $\mathcal{E}$RGO objects as specified in Section 10.4. These predicates handle XML, SGML, HTML, and XHTML, respectively.

parse_xpath_xml(`+?Source,+?XPath,+?NamespacePrefixList,-?Warn,+?Module`)**@flrxml**

parse_xpath_sgml(+?Source,+?XPath,+?NamespacePrefixList,-?Warn,+?Module)**@flrxml**

parse_xpath_html(+?Source,+?XPath,+?NamespacePrefixList,-?Warn,+?Module)**@flrxml**

parse_xpath_xhtml(+?Source,+?XPath,+?NamespacePrefixList,-?Warn,+?Module)**@flrxml**

The arguments have the following meaning:

`Source` specifies the input document; this parameter has the same format as in `load_structure`. `?XPath` is an XPath expression specified as a Prolog atom. *?Module* is the module where the resulting $\mathcal{E}$RGO objects should be placed. `?Module` must be bound. *?Warn* gets bound to a list of warnings, if any are generated during the processing, or to an empty list.

`?NamespacePrefixList` is a string that looks like a space separated list of items of the form *prefix = namespaceURL*. This allows one to use namespace prefixes in the XPath query given in the *?XPath* parameter. For example if the XPath expression is `'/x:html/x:head/x:meta'` where `x` stands for `'http://www.w3.org/1999/xhtml'`, then this prefix would have to be defined in `?NamespacePrefixList`:

```
parse_xpath_xhtml(url('http://w3.org'),
                  '/x:html/x:head/x:meta',
                  'x=http://www.w3.org/1999/xhtml',
                  ?Warnings,
                  foomodule)@flrxml.
```

10.4 Mapping XML to $\mathcal{E}$RGO

This mapping is based on a proposal by Guizhen Yang. It specifies how an XML parser can construct the corresponding F-logic objects as a result of parsing an input XML document.

The basic idea is as follows:

- Elements in XML are modeled as objects in F-logic.
- Subelements in XML are modeled as multivalued attributes in F-logic.
- Element attributes in XML are modeled as single-valued attributes in F-logic. This complies to the XML 1.0 specification which states that an attribute be defined only once for each element in an XML document.

This proposal deals with data-intensive XML documents, i.e., those that don't rely on the interpretation of comments or processing instructions to carry data. However, this proposal does consider the modeling of mixed element content in which text and subelements are interspersed.

We do not consider modeling XML entities either, assuming no entity references or that all entity references in the original XML document are already resolved by an XML parser.

10.4.1 Object Ids

According to the XML specification 1.0, an XML element can be defined with an oid that is unique across the document. Such an oid can be provided as the value of an element attribute of type ID, although this attribute can be arbitrarily named. Since an XML element is modeled as an F-logic object, we would like the oid of this object to take the value of any ID attribute if such value is defined. Otherwise, the oid must be automatically generated by the system.

Sitting on top of the XML root element, there is an additional root object which just functions as the access point to the entire object hierarchy.

Note:

- Only a validating XML parser can decide whether an attribute is an ID attribute since such definition is provided by a DTD.
- The oids of leaf nodes, which have no outgoing edges and carry plain text, are just the string values.

For example, the following XML document:

```
<?xml version="1.0"?>
<person ssn="111-22-3333">
  <name first="John"                          (Example 1)
        last="Smith"/>
</person>
```

is represented by the following F-logic objects, provided we already know that `ssn` is an ID attribute:

```
o1[person -> {'111-22-3333'}].
'111-22-3333'[ssn -> '111-22-3333',
               name -> {o2}
              ].
o2[first -> 'John', last -> 'Smith'].
```

10.4.2 Text and Mixed Element Content

The content of an XML element may consist of plain text, or subelements interspersed with plain text, for instance:

```
<greeting>Hi! My name is <first>John</first> <last>Smith</last>.</greeting>
```

Each text segment is modeled in F-logic as if it were referred to by a special tag, named `'$text'`. Corresponding to each text segment, a node is created and is referred to from the parent node by an edge labeled `'$text'`. The text becomes the value of a special single-valued attribute, named `'$string'`, of the newly created node. Moreover, if the content

of an XML element consists solely of plain text, then the text also becomes the value of a special single-valued attribute of this element, named `'$content'`, which is introduced for convenience purpose.

Therefore, the above XML segment would generate the following F-logic objects, where `o1, ..., o9` are new oids:

```
o1[greeting -> {o2}].
o2['$text' -> {o3},
   first -> {o4},
   '$text' -> {o6},
   last  -> {o7},
   '$text' -> {o9}
  ].
o3['$string' -> 'Hi! My name is '].
o4['$content' -> 'John', '$text' -> {o5}].
o5['$string' -> 'John'].
o6['$string' -> ' '].
o7['$content' -> 'Smith', '$text' -> {o8}].
o8['$string' -> 'Smith'}.
o9['$string' -> '.'].
```

Note:

- Handling of the whitespaces depends on the application. In the examples shown here, we assume that "insignificant" whitespaces have been omitted by the XML parser.
- `'$content'` can also be defined for every element. Its value is just the ASCII string as it appeared in the original document.

10.4.3 Multivalued XML Attributes

XML element attributes of type IDREFS are multivalued, in the sense that their value is a string consisting of one or more oids separated by whitespaces. Therefore, the value of such an attribute is a set. To stick to the convention that element attributes are modeled as single-valued attributes in F-logic, the value of an XML IDREFS attribute is represented as a list.

For example, the following XML segment:

```
<paper id="yk00" references="klw95 ckw91">
  <title>paper title</title>
</paper>
```

will generate the following F-logic atoms, assuming that the `reference` attribute is of type IDREFS:

```
yk00[references -> [klw95,ckw91],
```



```
      title -> {o1}
     ].
o1['$content' -> 'paper title', '$text' -> {o2}].
o2['$string' -> 'paper title'].
```

Note that since attribute definitions are provided by DTDs, only a validating XML parser can decide whether an element attribute has the type IDREFS. A non-validating parser should just output the attribute values as strings. The semantics of these strings is subject to further interpretation by the applications.

10.4.4 Ordering

XML is order-sensitive, since XML DTDs impose order among elements. For example, the following DTD element definition

```
<!ELEMENT book (author+, title, ISBN?)>
```

states that the content of a `book` element consists of one or more `author` element, followed by one `title` element, followed by an optional `ISBN` element. Note that XML 1.0 does not prescribe any order among element attributes.

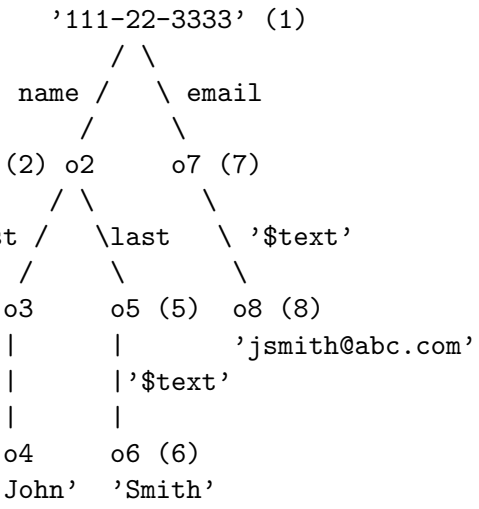
Preserving the order of elements can also be useful for translating F-logic objects back into an XML document.

A total order among the elements in an XML document should suffice to establish the order in which the elements appear sequentially in the XML document. In other words, such a total order should correspond to the order in which the element open tags appear. In addition, the ordering should also take into account the mixed element contents in which each text segment is referred to by a `'$text'` attribute.

```
<?xml version="1.0"?>
<person ssn="111-22-3333">
  <name>
    <first>John</first>                          (Example 2)
    <last>Smith</last>
  </name>
  <email>jsmith@abc.com</email>
</person>
```

For example, the XML document in Example 2 can be represented by the following tree, in which the integers enclosed by parentheses beside the nodes represent the order assigned to the elements:

```
                    o1 (0)
                    |
                    | person
                    |
```

Pre-order traversal of the XML tree will generate the total order for the XML elements and text segments.

The ordering information that exists in XML documents is encoded as follows. For each node, we introduce a special single-valued attribute, named `'$order'`, which stores the order number.

```
<bibliography>
<paper id="sb97">
  <author>
    <first>John</first>
    <last>Smith</last>
  </author>
  <author>
    <first>David</first>
    <last>Brown</last>
  </author>                                    (Example 3)
</paper>

<paper id="s91">
  <author>
    <first>John</first>
    <last>Smith</last>
  </author>
</paper>
</bibliography>
```

10.4.5 More on Special Attributes

The following special attributes are all single-valued. Note that this proposal does not intend to eliminate redundant information that may exist across various attributes. It is up to the implementation to decide whether to generate these attributes extensionally or intentionally.

1. `'$tag'`
   For each node, `'$tag'` returns the unordered label of the edge pointing to this node. `'$tag'` can be defined as follows:

   ```
   ?O['$tag' -> ?Tag] :- ?[Tag -> ?O].
   ```

   Note that for a node representing a text segment, the value of its `'$tag'` attribute is `'$text'`.

2. `'$parent'`
   For each node, `'$parent'` returns the oid of the parent node.

3. `'$leftSibling'`
   For each node, `'$leftSibling'` returns the oid of the node appearing immediately before the current node. This attribute is not defined for the nodes without a left sibling.

4. `'$rightSibling'`
   For each node, `'$rightSibling'` returns the oid of the node appearing immediately after the current node. This attribute is not defined for those nodes without a right sibling.

5. `'$childrenNum'`
   For each node, `'$childrenNum'` returns the number of children including nodes representing text segments.

6. `'$childrenList'`
   For each node, `'$childrenList'` returns a list, which is ordered, of the oids of its children. Note that each text segment is also counted as a child node.

7. `'$child'(N)`
   For each node, `'$child'(N)` returns the `N`-th child, where `1` $\leq$ `N` and `N` $\leq$ `'$chidlrenNum'`.

8. `'$tagList'`
   For each node, `'$tagList'` returns an ordered list of the tags of its children. Note that each text segment is also counted as if it were enclosed by a `'$text'` tag.

9. `'$tag'(N)`
   For each node, `'$tag'(N)` returns the tag of the `N`-th child, where `1` $\leq$ `N` and `N` $\leq$ `'$chidlrenNum'`. This attribute can be defined as follows:

   ```
   O['$tag'(N) -> Tag] :- O['$child'(N) -> V['$tag' -> Tag]].
   ```

Note: The aforesaid attribute `'$content'` can be defined for the nodes whose content is pure text as follows:

```
O['$content' -> String] :-
 O['$childrenNum' -> 1].text(1)['$string' -> String].
```


Bibliography

[1] H. Kyburg and C. Teng. *Uncertain Inference*. Cambridge University Press, 2001.