A Guide to $\mathcal{E}$RGO$\mathcal{AI}$ Packages

Version 2.0 (Myia)

Edited by

Michael Kifer
Coherent Knowledge

October 2018

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* (used most commonly), which enables Java programs to send arbitrary queries to $\mathcal{E}$RGO and get results, and an *experimental high-level API*, which is more limited and requires some setup, but can simplify a number of tasks in interfacing the two systems. 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* [1], 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).`".

The `FloraObject` object. While reading this document one will notice that the class `FloraObject` is used in many cases. This class consists of Java objects that encapsulate $\mathcal{E}$RGO objects. These Java objects are mostly used internally. From the end user's point of view, the only method of interest in this class is `toString()`.

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 two Java properties are set either as part of the java command (e.g., `java ...  -DPROLOGDIR=some-dir ...`) or inside the Java application itself, e.g.,

```
System.setProperty("PROLOGDIR","C:\\JSmith\\XSB\\config\\x64-pc-windows\\bin");
```

Please remember that Windows uses backslash as a file separator, and inside a Java program these backslashes must be doubled, as shown above.

The two aforementioned properties are:

PROLOGDIR: This variable points to the folder containing the XSB executable (binary, not the command script).
To get the right value for your installation, start ERGO and execute this at the prompt:

```
system{bindir = ?D}.
```

The result will be returned in the variable `?D`.

FLORADIR: This variable must point to the folder containing the ERGO installation.
To get the right value for your installation, start ERGO and execute this at the prompt:

```
system{installdir = ?D}.
```

Again, the result will be returned in the variable `?D`.

In order to be able to access ERGO, the Java program must first establish a session for a running instance of ERGO. 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 `net.sf.flora2.API.FloraSession`. This class provides methods for opening/closing sessions and loading ERGO knowledge bases (which are also used in the high-level interface). In addition, a session provides methods for executing arbitrary ERGO queries. The following is the complete list of the methods that are available in that class. All these are *public instance* methods and the word "public" is therefore omitted.

- `FloraSession()`

  This constructor creates a connection to an instance of ERGO. Use it like this:

  ```
  FloraSession session = new FloraSession();
  ```

  All the methods below are executed on `FloraSession`-objects produced in this way.

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

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

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

- Iterator<FloraObject> executeQuery(String query)

  This method executes the ERGO query given by the parameter query. The query must be terminated with a period, exactly as it would be typed in the ERGO shell. It is used to execute ERGO 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 net.sf.flora2.API.FloraSession. The examples below illustrate how to process the results returned by this method.

- Iterator<HashMap<String,FloraObject>> executeQuery(String query,Vector vars)

  This method executes the ERGO query given by the first argument. The query must be terminated with a period, as if it were typed in the ERGO shell. 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 ERGO 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 net.sf.flora2.API.FloraObject.

  The examples below show how to handle the results returned by this method.

- boolean executeCommand(String command)
  This is a simplified way of executing ERGO queries that *do not need* to return any results, i.e., when the user wants to know if the query is true or false for *some* bindings of command's arguments (if there are any), but not the actual bindings. There are also differences (compared to executeQuery()) in the way this command handles exceptions, as explained in the next section. As before, the command must be terminated with a period.

- boolean loadFile(String fileName,String moduleName)

  This method loads the ERGO program, specified by the parameter fileName into the ERGO module specified in moduleName. If errors occur during loading, loadFile() returns false.

- boolean compileFile(String fileName,String moduleName)

  This method compiles (but does not load) the ERGO program, specified by the parameter fileName for the ERGO module specified in moduleName. If errors occur during compilation, compileFile() returns false.

- boolean addFile(String fileName,String moduleName)

  This method adds the ERGO program, specified by the parameter fileName to an existing ERGO module specified in moduleName. If errors occur during addition, addFile() returns false.

- boolean compileaddFile(String fileName,String moduleName)

  This method compiles the ERGO program, specified by the parameter fileName for addition to the ERGO module specified in moduleName. If errors occur during compilation, compileaddFile() returns false.

The code snippet below illustrates the low-level API.

Step 1: Writing ERGO programs to be called by Java. 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 ERGO. The following code loads a ERGO 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
    if (session.loadFile(fileName,"basic_mod"))
        System.out.println("Example loaded successfully!");
    else
        System.out.println("Error loading the example!");

    /* 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);

    /* Printing 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-ℰRGO API, please see Section 1.4.

1.2 Debugging ℰRGO Statements Used in a Java Program

1.2.1 Logging

It often happens that an ℰRGO query or a file used from within a Java Program has an error and `executeQuery()` returns an error message saying that there is a "problem" with a ℰRGO statement. The Java part does not know what the problem is but it can be told to show the output of the ℰRGO statements on the console. This can be done by executing the statement

```
    FloraSession.showOutput();
```

This will stream all warnings, errors, and just normal output from ℰRGO to the console. When no longer needed, this mode can be turned off like this:

```
    FloraSession.hideOutput();
```

The `FloraSession.showOutput();` statement can be executed at any point in the program, but preferably after calling `FloraSession()`, to avoid irrelevant output.

Here is an example of what you might see:

```
My command 2 has succeeded

yes

Loading file test.ergo:

yes
```

```
++Error[Ergo]> [test.ergo] <Composer> near line(2)/char(3) ‘b’ unexpected operand:
 ’,’, ’.’, or some other operator may be missing just before the indicated location

++1 error

++compilation aborted
```

Here the output from ERGO starts with normal output from, say, `writeln(...)@\io` commands and ends with a compilation error encountered while compiling the file `test.ergo`. (The above is output from ERGO; in FLORA-2 it will be similar.)

Two other useful calls are

```
FloraSession.enableLogging();
FloraSession.disableLogging();
```

Executing `FloraSession.enableLogging();` will cause the Java API to record all major events such as starting a session, ending it, loading or adding a ERGO file, and execution of every query. It does not report query answers, however. Logging can be enabled or disabled anywhere in the program, but, of course, the log messages will start to appear only after the enabling command is executed and will cease to appear after `disableLogging()` is executed.

1.2.2 Catching Exceptions, Checking Errors and Warnings

Logging allows one to find errors and warnings in an ERGO subprocess of a Java program by checking the output produced by the session. However, often one needs to be able to do this *programmatically* and this can be done as follows.

Exceptions. First, if a running ERGO query invoked via `executeQuery()` issues a run-time error, a Java `FlrException` is thrown, and this can be caught by the parent Java program. An exception is also thrown if the query passed to `executeQuery()` has a syntax error.[1] Note that ERGO statements invoked via the other methods (`executeCommand()`, `loadFile()`, etc.) do *not* throw exceptions and errors that occur during the execution of those statements can be detected only through the mechanism of `hasErrors()` described below. However, `executeCommand()`, `loadFile()`, `addFile()`, etc., may throw exceptions for other reasons, such as a wrong module in `loadFule()`, etc.

Detecting syntax errors and warnings. Errors and warnings produced by the ERGO compiler do *not* result in exceptions, so they cannot be caught via Java's try-catch mechanism. To detect if a previous `executeQuery()`, `executeCommand()`, `loadFile()`, or similar API command produced a warning or an error, use the following *public instance* methods in class `FloraSession`:

[1] If the query statement itself has no syntax error but it loads a file that has a syntax error then *no* exception is thrown. Read on to see how to identify this situation programmatically.

- `boolean hasErrors()` — executing `session.hasErrors()`, where session is a variable holding a `FloraSession` object created earlier, will tell if a previous `executeQuery()` or other such command (executed on the same `FloraSession` object) has produced an error. This also includes runtime errors, like `FlrException`'s.
- `boolean hasWarnings()` — executing `session.hasWarnings()`, where session is a variable holding a `FloraSession` object, will tell if a previous `executeQuery()` or other such command (executed on the same `FloraSession` object) has produced a warning.

1.3 The High-Level Interface (experimental)

The high-level API operates by creating proxy Java classes for ℰRGO classes selected by the user. This enables the Java program to operate on ℰRGO classes by executing appropriate methods on the corresponding proxy Java classes. However, compared to the low-level interface, the high-level one is somewhat limited. Both interfaces can be used at the same time, if desired.

Note A: Most users appear to opt for the low-level interface and such readers can skip this section.

Note B: This interface will not work for ℰRGO 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.

The use of the high-level API involves a number of steps, as described below.

Stage 1: Writing ℰRGO programs to use with the high-level interface. We assume the same `flogic_basics.flr` file as in the previous example.

Stage 2: Generating Java classes that serve as proxies for ℰRGO classes. The ℰRGO side of the Java-to-ℰRGO high level API provides a predicate to generate Java proxy classes for each F-logic class which have a signature declaration in the ℰRGO 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 ERGO methods and the `getVDN_*`, `setVDN_*`, etc., forms are for dealing with non-inheritable ERGO 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 ℰRGO query (defined in the `javaAPI` package in ℰRGO). 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 ℰRGO 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 ℰRGO). 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 ℰRGO class `person`. The parameters for the "get", "set" and "delete" Java methods are the same as for the corresponding ℰRGO methods. The first constructor for class `person` takes a low-level object of class `net.sf.flora2.API.FloraObject` as a parameter. The second parameter is the ℰRGO 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 ℰRGO 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 ℰRGO 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 ERGO 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 ERGO objects. These Java objects are members of the class `net.sf.flora2.API.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 */
  if (session.loadFile(fileName,"basic_mod"))
      System.out.println("Example loaded successfully!");
  else
      System.out.println("Error loading the example!");

  // 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();
  }
}
```

1.4 Executing Java Application Programs that Call ℰRGO

To compile and run Java programs that interface with ℰRGO, follow the following guidelines.

- *Compilation*: 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 classpath using the `-classpath` parameter (the first line is for Windows and the second for Mac and Linux):

  ```
  -classpath "%FLORADIR%\java\flora2java.jar";"%FLORADIR%\java\interprolog.jar"
  -classpath "$FLORADIR/java/flora2java.jar":"$FLORADIR/java/interprolog.jar"
  ```

  `FLORADIR` here is a shell (cmd, in Windows) variable that can be set by the scripts `flora_settings.sh` (Linux/Mac) or `flora_settings.bat` (Windows). In sum, the Java compilation command should look as below, where `JAVA_BIN` is a shell/cmd variable that points to the directory containing the Java compiler command (again, the first command below is for Windows and the second for Linux/Mac):

  ```
  %JAVA_BIN%\javac -classpath
       "%FLORADIR%\java\flora2java.jar";"%FLORADIR%\java\interprolog.jar"
  $JAVA_BIN/javac -classpath
       "$FLORADIR/java/flora2java.jar":"$FLORADIR/java/interprolog.jar"
  ```

  Note: the above commands should each be on one line.

- *Running*: Generally, Java programs that call ℰRGO should be invoked using the following command. For Linux and Mac, change *%VAR%* to *$VAR*:

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

  The above commands use several shell/cmd variables that are explained below. Instead of using the variables, one can substitute their values directly—read on.

  - `JAVA_BIN`: This variable should point to the directory containing the `java` and `javac` executables of the JDK. It can be set by executing the scripts `Ergo\java\windowsVariables.bat` or `Ergo/java/unixVariables.sh`, depending on your OS.

    Of course, if the `javac` and `java` commands can be found through the `PATH` environment variable then one can simply type `javac` and `java` instead of the above.
  - `PROLOGDIR`: This variable should point to the directory containing the XSB executable, which can be accomplished by executing the scripts `Ergo\java\flora_settings.bat` (Windows) or `Ergo/java/flora_settings.sh` (Linux/Mac).
    Alternatively, one can get the correct value of `%PROLOGDIR%` (or `$PROLOGDIR`) and use it directly in the `-DPROLOGDIR=...` option by starting ℰRGO and executing this query at the prompt:

    ```
    system{bindir = ?D}.
    ```

    The result will be returned in the variable `?D`.
  - `FLORADIR`: This variable should be set to the directory containing the ℰRGO system, which can be done by executing the aforesaid scripts `flora_settings.bat` and `flora_settings.sh`.
    Alternatively, one can get the correct value of `%FLORADIR%` (or `$FLORADIR`) and use it directly in the `-DFLORADIR=...` option by starting ℰRGO and executing this query at the prompt:

    ```
    system{installdir = ?D}.
    ```

    Again, the result will be returned in the variable `?D`.
  - `MYCLASSPATH`: This variable should include the correct paths to the jar files containing the API code, i.e., `flora2java.jar` and file `interprolog.jar`, plus the directory where the main application class (like `flogicbasicsExample` in our example) is found. Normally, one sets `MYCLASSPATH` to `%CLASSPATH%;%FLORADIR%\java\flora2java.jar;%FLORADIR%\java\interprolog.jar;DirOfTheExample`, where `DirOfTheExample` is the directory where the main application class resides. In our example, this directory is simply . (the current directory). For Linux and Mac, use ':' instead of ';' as a separator, forward slashes instead of backward ones, and *$VAR* instead of *%VAR%*.

One can, of course, substitute the contents of the MYCLASSPATH variable directly into the -classpath %MYCLASSPATH% part of the above Java/Javac invocation commands.

  – The variable java.library.path in the above command is optional. It needs to be set *only if XSB is configured to use the native Java interface* (which usually is not the case).

- Some Java applications may employ additional Java properties. 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 property INPUT_FILE to be set with the -D option at the Java invocation time. In general, such additional properties can be also set via the method System.setProperty() inside the Java application. In our particular case, the program expects that INPUT_FILE is set to point to the flogic_basics.flr ERGO file, which it then loads. In other words, the java command shown above also needs this parameter:

```
-DINPUT_FILE="%INPUT_FILE%"    (Windows)
-DINPUT_FILE="$INPUT_FILE"     (Linux/Mac)
```

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

1.5 How Do Applications Find the Knowledge Base?

When a Java application starts ERGO, the latter determines the default runtime directory in which it will work. Usually, this is the directory in which your Java application runs. You can find out which directory it is by sending the following query to ERGO:

```
File[cwd->?Dir]@\io.
```

?Dir will be bound to the runtime directory and Java can get that value as explained earlier. Your Java application can change that directory via this query:

```
File[chdir('....new current dir...')]@\io.
```

The simplest basic rule is that all ERGO's files that your Java application loads, adds, etc., must be specified relative to the current directory.

One can also put additional directories to the ERGO's search path by executing the query

```
Libpath[add('....new dir to search...')]@\sys.
```

Then your application can use file names not only relative to the runtime directory but also relative to any of the directories added in this way. Note that this may put many directories on the search path, and several of them may have similarly named files. Therefore, one must make sure that the search is unambiguous.

1.6 Summary of the Variables and Properties Used by the Interface

The Java-ℰRGO interface needs the following variables and properties to be set:

- `JAVA_HOME` – this is an OS environment variable. It is normally set when you install Java. Normally, Java will not work correctly if this environment variable is not set correctly.
- The following Java properties must be set for the Java API to work. They can be set either through the `-D` option of the `java` command or inside the Java application via `System.setProperty("propertyname",value)`.
  - `FLORADIR` — the path to the ℰRGO installation directory.
  - `PROLOGDIR` — the path to the folder containing XSB executable.

  The proper values for these properties can be obtained from ℰRGO by running these queries, respectively:

  ```
  system{installdir=?D}.
  system{bindir=?D}.
  ```

- The following shell/cmd variable may need to be set, as explained above, if the `java` and `javac` commands cannot be found through the `PATH` environment variable of your OS. It can be set by the scripts `unixVariables.sh` or `windowsVariables.bat`, whichever applies to your OS:
  - `JAVA_BIN` — the directory where Java executables `java` and `javac` live. It is usually set to `$JAVA_HOME/bin` or `%JAVA_HOME%\bin`, depending on the OS.

1.7 Building the Prepackaged Examples

Sample applications of the Java-ℰRGO interface are found in the `java/API/examples` folder of the ℰRGO distribution. To build 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 `java/API/examples`. For instance, to run the `flogicbasicsExample`, use this command on Linux and Mac:

```
runExample.sh flogicbasicsExample
```

On Windows, use this:

```
runExample.bat flogicbasicsExample
```

Chapter 2

ERGO-to-Java Interface: Calling Java from ERGO

by Michael Kifer

This chapter describes the API for opening some standard Java widgets from within ERGO rules. This API also allows one to call arbitrary Java programs and thereby use ERGO for scripting Java applications.

The ERGO-to-Java API works both when ERGO 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 ERGO-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]@\e2j` - the variable `?Mode` will be bound to one of the following:
  - `studio` – if ERGO runs as part of Ergo Studio.
  - `[ergo2java,gui]` – if ERGO runs as a standalone mode in an environment that supports graphics. This is usually the case when one invokes ERGO in a command window on a personal computer.
  - `[ergo2java,nogui]` – this is usually the case when ERGO 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]@\e2j` – 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 ERGO (e.g., via the Task Manager or System Monitor). Java is also killed when `\end` is executed at the ERGO prompt.

- `System[path(studioLogFile)->?File]@\e2j` – 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 find out which button was clicked by the user. Several types of dialog boxes are supported:

- `Dialog[show(?Question)->?Answer]@\e2j` – 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]@\e2j` – 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)]@\e2j` – 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]@\e2j` – pops up a file chooser. `?File` gets bound to the file chosen by the user.

- `Dialog[chooseFile(?ExtensionsList)->?File]@\e2j` – 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]@\e2j` – 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)]@\e2j` – 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)]@\e2j` – closes the specified window.
- `Window[alive(?Window)]@\e2j` – 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)]@\e2j` – erases the contents of the given window.
- `Window[print(?Window,?Text)]@\e2j` – 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)]@\e2j` – load the specified jar-file into the system.

- *JavaObjSpec*`[message(`*JavaMethodWithArgs*`) ->` *Result*`]@\e2j` – invoke Java method on a Java object and return the result.
  This feature is *experimental* and incomplete. It does not support all Java data structures and not all kinds of methods can be applied.

JavaObjSpec in the above `message(...)` API can have several forms:

- `oid(`*Integer*`)`: When Java returns an object, it is registered by ERGO and is represented by an integer, e.g., 345. In order to invoke a Java method on it, the *JavaObjSpec* must be specified as `oid(345)`.

- If *JavaObjSpec* is an integer, float, or atom in ERGO then it is interpreted as a Java long, double, or string, respectively. Java methods that apply can be used in `JavaMethodWithArgs`. For instance,

```
?- '123abc789'[message(split(abc))->?P]@\e2j.
?P = ['123', '789']
?- abc23op[message(matches('.+23.*'))->?P]@\e2j.
?P = \true
?- abcdf[message(getBytes)->?R]@\e2j.
?R = "abcdf"^^\charlist
```

- If *JavaObjSpec* is a list, it is interpreted as an array of objects. For instance, the list `[pp,i(8),k(u,m)]` is mapped into a Java array and `[1]` is a method applied to that array, which returns the second element.

```
?- [pp,i(8),k(u,m)][message([1])->?P]@\e2j.
?P = i(8)
?- [pp,i(8),k(u,m)][message(length)->?P]@\e2j.
?P = 3
```

- If *JavaObjSpec* is `byte(`*smallNumber*`)`, `short(`*shortinteger*`)`, or `int(`*integer*`)` then it is interpreted as a byte or int constant.

- If *JavaObjSpec* is `term(`*someHiLogTerm*`)` then *someHiLogTerm* is mapped into a `TermModel` object—see `http://interprolog.com/ipjavadoc/com/declarativa/interprolog/TermModel.html` for the details of this class, which has methods to de/compose terms on the Java side.

```
?- term(p(a,b))[message(getFunctorArity)->?P]@\e2j.
?P = 'p/2'
```

- If *JavaObjSpec* is of the form `array(`*type,list*`)` then this is mapped to an array of constants of the given type (string, byte, int, float). For instance

```
?- array(int,[2,7,99])[message([2])->?P]@\e2j.
?P = 99
```

- If *JavaObjSpec* is of the form *Wrap(const)*, where *Wrap* is one of Boolean, Character, Byte, Double, Float, Integer, Long, or Short and *const* is of the appropriate ERGO type, then this will be mapped into an object of type `java.lang.Boolean`, `java.lang.Character`, etc., respectively. Note that these are objects, while the wrappers `int`, `short`, `byte`, etc., which we introduced earlier are non-object constants.

- Additionally, to be able to execute *static methods*, fully-qualified classes wrapped with `class` can be used. For instance,

```
?- class('java.lang.String')[message(format('abc=%d %s',12,iiii))->?R]@\e2j.
?R = 'abc=12 iiii'
?- class('java.lang.String')
              [message(String(array(byte,[119,111,114,108,100])))->?P]@\e2j.
?P = world
```

- It is further possible to send a message to a *static variable* (i.e., invoke a method on the object held by that static variable, as in `java.lang.System.out.println("Hello")`) in a class as follows:

```
?- class('java.lang.System'+out)[message(println(Hello))->?R]@\e2j.
Hello
?R = \@?   // a null value because println returns void
?- class('java.lang.System'+out)[message(printf('%1.16g', 69.1))->?R]@\e2j.
69.10000000000000
?R = oid(1)
```

  Note that here `out` is the static variable in class `java.lang.String` to which the messages are being sent.

  Currently it is not possible to get the value of a static variable unless there is a getter-method for that variable.

The same mapping conventions are applied to the arguments of the method-expressions passed to `message(...)` in our API call.

Chapter 3

pyergo: A Python-to-$\mathcal{E}$RGO Interface

by Michael Kifer

This interface allows Python programs to start $\mathcal{E}$RGO, load knowledge bases into it, and then query and modify them. One can also talk directly to the underlying Prolog engine, XSB. This API works both with Python 2.7 and 3+.

3.1 Introduction

The *pyergo* interface consists of four types of APIs: one for starting and closing $\mathcal{E}$RGO sessions, one for querying $\mathcal{E}$RGO, one for talking directly to XSB, and one for parsing the results. A fairly extensive example of a program that does both is found in

```
.../Ergo/python/pyergo_example.py
```

in the $\mathcal{E}$RGO distribution. This program provides several examples of using the `pyergo` interface, including various edge cases and exception handling. The easiest way to try these examples is via the provided shell scripts,

```
.../Ergo/python/runpyergo.sh    -- Linux/Mac
.../Ergo/python/runpyergo.bat   -- Windows
```

One only has to change the two variables in those scripts `ERGOROOT` and `XSBARCHDIR`.

To make it possible for your Python program find $\mathcal{E}$RGO, two parameters are to be provided: the architecture directory, called `XSBARCHDIR` in those scripts and in `pyergo_example.py`, and the root directory for the $\mathcal{E}$RGO reasoner, which we called `ERGOROOT`. The names of these variables are, of course immaterial, but we will use these names here for easy reference. Pay attention to how this information is passed from the scripts to the program via the `sys.argv` array, as this is one of the most convenient methods.

How will you know what to substitute for the aforesaid ERGOROOT and XSBARCHDIR? This is easy! Just start ERGO and type these two queries:

```
?- system{installdir=?Ins}.  // gives ERGOROOT
?- system{archdir=?Arch}.    // yields XSBARCHDIR
```

The first query will give you ERGOROOT and the second XSBARCHDIR.

Next your program needs to be told where the interface can be found. In our example, this is accomplished via

```
import sys
sys.path.append(ERGOROOT.replace('\\','/') + '/python')
```

Of course, you will have to replace ERGOROOT here appropriately, as explained. Note that forward slashes are preferred, although backward slashes are also recognized in Windows (sometimes they must be escaped with another backslash to satisfy Python syntax). Finally, import the API as follows:

```
from pyergo import \
    pyergo_start_session, pyergo_end_session, \  to start/end Ergo session
    pyergo_command, pyergo_query,             \  to talk to Ergo
    HILOGFunctor, PROLOGFunctor,              \  to parse results from Ergo
    ERGOVariable, ERGOString, ERGOIRI, ERGOSymbol, \
    ERGOIRI, ERGOCharlist, ERGODatetime,      \
    ERGODuration, ERGOUserDatatype,           \
    pyxsb_query, pyxsb_command,               \  to talk to XSB directly
    XSBFunctor, XSBVariable, XSBAtom,         \  to parse results from XSB
    XSBString,                                \
    PYERGOException, PYXSBException              to process exceptions
```

In most cases you will need only a small subset of these functions, but importing them all is easy, does not hurt, and is useful in case you later extend your program. (Of course, delete the blue comments.)

3.2 Connecting to ERGO

This part of the API consists of two commands: pyergo_start_session() — to start ERGO and connect to it, and pyergo_end_session() — to unload ERGO.

- pyergo_start_session(): This takes two arguments, the aforementioned directories:

  ```
  pyergo_start_session(XSBARCHDIR,ERGOROOT)
  ```

  Do not forget to replace XSBARCHDIR and ERGOROOT, as discussed. Errors will be thrown if one of these directories is missing, unreadable, or does not look like belonging to a valid ERGO installation.

- `pyergo_end_session()`: Used to unload ERGO. For example,

  ```
  pyergo_end_session()
  ```

  This command is not needed if your program exits soon after unloading, but it can save resources, if your Python program uses ERGO in the beginning only and then continues to work for a significant period of time till the end, without accessing the knowledge base.

3.3 Talking to ERGO

This part of the API consists of two commands also: `pyergo_command()` and `pyergo_query()`. The difference is that the first is not expected to return any results and exception is thrown if the command returns *False*. It also throws exceptions if something went wrong during the compilation or execution of the command. In contrast, `pyergo_query()` throws far fewer exceptions.

- `pyergo_command()`: Execute an ERGO command passed as a parameter. Takes a query and treats it as a command, i.e., ignores the results and expects it to succeed. This is usually used to load a file, do something that does not return results (e.g., insert facts or rules), and when the command returns *False* then it is treated as an error so an exception is raised. For example,

  ```
  pyergo_command("writeln(Ergo = 'aaaa bbb')@\\io.")
  pyergo_command('insert{qq({11,fff(22)},33)}.')
  pyergo_command("\\false.")  # will throw an error
  ```

  Note that whenever ERGO requires backslashes, they must be doubled and the commands must end with a period, as usual.

- `pyergo_query()`: Execute a query passed as a parameter and get results. Like `pyergo_command()`, it takes an ERGO query, but that query may or may not have results and when it does the results are returned. The results are returned as an array of 4-tuples, which can be iterated over, as explained in the example below. Note: `pyergo_command()` never throws an exception that has to do with the compilation or execution of a query. Instead, it returns that exception information as part of the result. The only exception it is supposed to throw is when a query returns an unsupported variable binding, i.e., something that is not a Prolog or HiLog term (like, for example, a reified formula).

Here is how one gets results from a query. Suppose `qq/1` was previously assigned the tuples (11, 33) and (fff(22), 33). Then:

```
for row in pyergo_query('qq(?X,?Y).  '):
    print("result: ",row[0],row[1],row[2],row[3].value)
```

will print (except the top line, which is added for readability):

```
# Result                      Compile Status      Truth Value    Exception
[('?X',11),('?Y',33)]    ('not_eof','success')    True            normal
[('?X',HILOGFunctor(name=fff,args=[22])),('?Y',33)]
                         ('not_eof','success')    True            normal
```

Here the compile status says whether the query was compiled without errors and whether an end of file (or string) was reached. In our case, it was not (`not_eof`) because the query string has some white space after the period, but usually it says `eof`. The truth value can be *True*, *False*, or *None*—the latter standing for "undefined" in ERGO terms. Finally, exception is `normal`, if no runtime exception happened in the query execution, and the actual exception is shown otherwise.

The most important component is the first, `row[0]`. It is a list of variable name/binding pairs, where the variable names are taken from the query (`'?X'` and `'?Y'` in our case). These are the actual query answers. If a query has no output variables, but the query is true, an empty list is returned. Silent variables (the ones that start with an underscore) are not returned. Note that the results returned can be complex terms (they are always returned as Prolog, not HiLog terms), like the Python object `HILOGFunctor(name=fff,args=[22])` in our case. We will provide the details of that part of the API in Section 3.5, but here is an example of how to unpack such objects:

```
for row in pyergo_query('qq(?X,?Y).  '):
    [(XVarname,XVarVal),(YVarname,YVarVal)] = row[0]
    if isinstance(XVarVal,HILOGFunctor):
        #Xresult=XVarname+'='+str(XVarVal.name)+' '+str(XVarVal.args)
        Xresult=XVarname+'='+str(XVarVal)
    elif isinstance(XVarVal,PROLOGFunctor):
        #Xresult=XVarname+'='+str(XVarVal.name)+' '+str(XVarVal.args)+'@\\plg'
        Xresult=XVarname+'='+str(XVarVal)
    else:
        Xresult=XVarname+'='+str(XVarVal)
    if isinstance(YVarVal,HILOGFunctor):
        #Yresult=YVarname+'='+str(YVarVal.name)+' '+str(YVarVal.args)
        Yresult=YVarname+'='+str(YVarVal)
    elif isinstance(YVarVal,PROLOGFunctor):
        #Yresult=YVarname+'='+str(YVarVal.name)+' '+str(YVarVal.args)+'@\\plg'
        Yresult=YVarname+'='+str(YVarVal)
    else:
        Yresult=YVarname+'='+str(YVarVal)
    print("result: ",Xresult+" and "+Yresult,row[1],row[2],row[3].value)
```

The commented out parts of this example show how to access various parts of the complex answers returned as HiLog or Prolog terms.

Note: only elementary data types, Prolog, and HiLog terms can be returned from Ergo to Python. More complex things, like reified predicates and frames, cannot be returned.

The commands `pyergo_query()` and `pyergo_command()` raise Python exceptions, if bad things happen during the execution. These exceptions are Python objects of the form

```
PYERGOException(query=..., command=..., message=...)
```

Some of the components may be missing in specific cases (e.g., `query` in case of a command, and vice versa).

3.4 Talking to XSB

This part of the API is **for expert users only** whose applications require talking to the underlying XSB engine directly. It consists of the commands `pyxsb_command()` and `pyxsb_query()`. The first is like `pyergo_command()` except that Prolog syntax is used for the query. The second, `pyxsb_query()`, differs more. Like `pyergo_query()`, it returns an iterable array of tuples, but the number of components in those tuples is arbitrary and each element corresponds to a variable binding in the query. The bindings are listed in the lexical order of appearance of the variables in the query. Silent variables are treated as any other variable, duplicate occurrences of the same variable are omitted, and the names of the variables are not made available. Likewise, compilation information is not returned and neither is the truth value, so it is not easily possible to tell whether a query is true or undefined. For example,

```
for row in pyxsb_query("p(X,Y,Y,_W).") :
        print(row[0],row[1],row[2])
```

Here `row[0]` corresponds to `X`, `row[1]` to `Y`, and `row[2]` to `_W` (recall that silent variables are *not* ignored in the XSB interface). True and undefined answers to the query will be printed without distinction. To separate these two types of answers, the XSB predicate `call_tv/2` can be used (read about it in the XSB manual).

Like with ℰRGO queries, the bindings (`row[0]`, `row[1]`, etc.) can be complex terms and variables. Unpacking that information is the subject of the next section.

One last difference is that `pyxsb_query()` and `pyxsb_command()` raise Python exceptions, if bad things happen during the execution. The exceptions are Python objects of the form

```
PYXSBException(code=..., query=..., command=..., type=..., message=...)
```

Some of the components may be missing in specific cases (e.g., `query` in case of a command, and vice versa).

3.5 Unpacking the Results

Unpacking the results returned by ℰRGO and XSB is conceptually similar. For integers, floats, and lists, both ℰRGO and XSB use the same native Python classes. However, for more complex data structures, `pyergo_query()` and `pyxsb_query()` use different Python classes. ℰRGO has more data types than XSB and thus needs more classes to represent them, so we discuss these issues separately.

3.5.1 Unpacking Results from pyergo_query()

- `ERGOSymbol(value=`*string*`)`: this represents ERGO abstract symbols (`"..."^^\symbol`). To get the actual string out of an `XSBAtom`-object, just use *obj*`.value`.
- `ERGOString(value=`*string*`)`: this represents the ERGO datatype `"..."^^\string`.
- `ERGOCharlist(value=`*string*`)`: this represents lists or Unicode characters and corresponds to the ERGO datatype `\charlist`.
- `ERGOIRI(value=`*string*`)`: this type of an object comes from an ERGO `\iri` literal.
- `ERGODatetime(date=`*date-list*`,time=`*time-list*`)`. A Python object of this form would come from an ERGO `\datetime` literal. For instance, the ERGO `\datetime`-literal `"2008-6-27T10:30:55.23456-0:20"^^\datetime` will give rise to `ERGODatetime(date=[1,2008,6,27],time=[10,30,55.23456,-1,0,20])`. In a date-list like `[1,2008,6,27]`, 1 means the year is CE and -1 means BCE. The rest stands for the year, month, and day. In a time-list, the elements are hours, minutes, seconds, UTC offset sign (1 or -1), UTC offset hour, and UTC offset minutes. Seconds and milliseconds are represented together using one positive decimal number.

  A `\time` object from ERGO would give rise to an `ERGODatetime` object in Python in which the `date` component is absent. A `\date` object in ERGO would give rise to a `ERGODatetime` object in which the `time`-component is absent.
- `ERGODuration(value=`*duration-list*`)`. This type of a Python object corresponds to an ERGO `\duration`-literal. For instance, `"-P22Y2M10DT1H2M3.0S"^^\duration` would give rise to a Python object of the form `ERGODuration(value=[-1,22,2,10,1,2,3.0])`. The components of the *duration-list* are sign (of the duration, 1 or -1), years, months, days, hours, minutes, and seconds (which is a decimal number that represents both seconds and milliseconds).
- `ERGOVariable(name=`*string*`)`: this type of objects may be returned if query results contain unbound variables. Note that the actual names of these variables are immaterial and they are almost always different from what was in the query. The only thing that matters is the equality among these names. For instance, if a tuple of bindings like `(ERGOVariable(name='_Var123'), ERGOSymbol(value='abc'), ERGOVariable(name='_Var123'))` is returned, it means that the first and the last components in that answer are the same variable.
- `PROLOGFunctor(name=`*string*, `args=`*list*, `module=`*xsb-module*`)` and `HILOGFunctor(name=`*string*, `args=`*list*`)`: these classes are used to represent Prolog and HiLog terms, respectively.

Typically, unpacking of an answer takes the form or testing what kind of object it is (e.g., `isinstance(obj,ERGOVariable)`, `isinstance(obj,ERGOString)`, `isinstance(obj,ERGOIRI)`, `isinstance(obj,HILOGFunctor)`, `isinstance(obj,int)`) and then proceeding to extract the relevant attributes of the object. An example of this was shown in Section 3.3.

3.5.2 Unpacking Results from pyxsb_query()

The function `pyxsb_query()` uses these classes for the data types it returns:

- `XSBAtom(name=`*string*`)`: this represents XSB atoms (i.e., ℰRGO symbols). To get the actual atom out of an `XSBAtom`-object, just use *obj*`.name`.
- `XSBString(value=`*string*`)`: this represents lists or Unicode characters. This class is provided for easier readability: this data type does not really exists in XSB in its own right.
- `XSBVariable(name=`*string*`)`: this type of objects may be returned if query results contain unbound variables. Note that the actual names of these variables are immaterial and they are almost always different from what was in the query. The only thing that matters is the equality among these names. For instance, if a tuple of bindings like `(XSBVariable(name='_Var123'), XSBAtom(name='abc'), XSBVariable(name='_Var123'))` is returned, it means that the first and the last components in that answer are the same variable.
- `XSBFunctor(name=`*string*`,args=`*list*`,module=`*xsb-module*`)`: this represents a complex term with the functor name *string* and arguments *lists*. The elements of the list can be terms, atoms, variables, etc. `XSBFunctor` objects is used only by `pyxsb_query()`.

Typically, unpacking of an answer takes the form or testing what kind of object it is (e.g., `isinstance(obj,XSBVariable)`, `isinstance(obj,XSBString)`, `isinstance(obj,XSBFunctor)`, `isinstance(obj,HILOGFunctor)`, `isinstance(obj,int)`) and then proceeding to extract the relevant attributes of the object. An example of this was shown in Section 3.3.

Chapter 4

HTTP and Web Services

by Michael Kifer

This chapter describes the API for issuing HTTP requests to Web servers. This facility could be used for reading and querying Web resources and, perhaps more importantly, for talking to Web services.

4.1 General

The $\mathcal{E}$RGO-to-Java API is available in the system module `\http` 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@\http`.

4.2 The HTTP API

The most important call in the $\mathcal{E}$RGO Web API is `http(...)`, described next.

- `?URL[http->(?Result,?Warnings)]@\http` — a basic request to bring back a Web page or to invoke a RESTfull service via a GET HTTP method. The result from the server is bound to `?Result` and the errors/warnings from the server, if any, to `?Warnings`. A result is an atom, which typically is in the HTML, XML, or JSON format. The warnings are represented as lists of atoms (one per warning) or as an empty list, if no warnings.

  If `?Result` is a zero-length atom, it means that the request failed for various reasons. Such reasons may or may not be explained as a warning—depending on the server.

- `?URL[http(?OptionList)->(?Result,?Warnings)]@\http` — a more complex request to a server, which specifies the requirements by passing a list of options. This API call supports GET, POST, PUT, and DELETE HTTP requests and can be used both for RESTfull as well as non-RESTfull Web services. The option list has this form:

`[` *option1, option2, ..., optionN* `]`

where each option either has the form *optionName* = *value* or is a Boolean option of the form *optionName*. (Currently there is only one Boolean option: `delete`.) No *optionName* can occur in the list more than once, or an error is issued. The following options are supported:

- `redirect`: the value must be `true` (default) or `false`. Tells the server whether to follow redirection or not.

  ```
  ?- 'https://google.com'[http([redirect=false])->?R]@\http.
  ```

  This will respond with an HTML document saying "The document has moved."
- `secure`: the value must be `false` (default) or a path-name to a local file, which contains certificates. The certificates must be in the PEM format `https://support.ssl.com/index.php?/Knowledgebase/Article/View/19/0/der-vs-crt-vs-cer` Such files typically have the `.pem` or `.crt` extension.

  If a file-path is specified, the server is verified with respect to the certificate. If unsuccessful, `?Result` is a zero-length atom.
- `timeout`: the value is a positive integer specifying the number of seconds to wait before aborting the request.
- `useragent`: the name of the user agent to use in the HTTP header when handshaking with the server. Some servers require this, but most do not. Servers have no way of verifying the user agent field. Example:

  ```
  ?- 'http://myurl.my'[http([timeout=7, useragent='My Ergo crawler'])
                                -> (?Res,?Warn)]@\http.
  ```
- `header`: The value is either an atom (if just one header needs to be passed) or a list of atoms, to specify several headers at once. Examples:

  ```
  header='User-Agent: just me'
  header=['Content-Type: application/json','Authorization: Bearer abcdefg']
  ```
- `auth`: the value must be user/password. This is used if the web site requires authentication. For example,

  ```
  auth=justme/mypasswd
  ```
- `post`, `put`: the value is an atom, typically in the JSON or XML format.

  These options, if given, will contact the server using the HTTP methods POST or PUT, respectively. If none of these options is given (and no `delete` option), the GET method is used. One cannot specify both of these at once, and the HTTP method must match what the server expects. In case of a mismatch, the server may (or may not) send an error message back, which would then be available in the warnings list mentioned above, or as contents in `?Result`. Example:

  ```
  ?- 'https://myurl.my'[http([auth='me@my'/'my+pw',
                                 post='{"message": "Hello"}'])->?R]@\http.
  ```
- `delete`: this is the only Boolean option, so it is specified just as `delete`. It tells the server to use the DELETE HTTP method. This option cannot be used together with `post` or `put` options. Example:

```
?- "https://google.com"^^\iri[http([delete])->?R]@\http.
```

(will respond with an HTML file saying "The request method DELETE is inappropriate for the URL"). This example also illustrates the point that the URL can be given as a plain symbol or as an `\iri` data type.

4.3 Miscellaneous

This package includes a number of other useful calls that are often used together with the `http(...)` method.

- `?URL[properties->?Props]@\http` — returns a list of properties of `?URL`: `[`*PageSize*,*ModTime*,*RedirectedURL*`]`. Here *PageSize* is the size of the page in bytes, and *ModTime* is the last modification time expressed as the number of seconds since epoch (1900-01-01). Some servers might not return one or both of the last two parameters, in which case -1 is returned. The last component is the actual URL of the page. If the page at `?URL` was not redirected then *RedirectedURL* is the same as the original URL. If that page has a redirection then this and all the intermediate redirections are followed and *RedirectedURL* is the final URL in that chain.

  This method contacts the network, but it is lighter than `http(...)` and it retrieves properties only.

  Example:

  ```
  ?- \"http://expedia.com"[properties->?R]@\http.
  ?R = [124787, 1509446769, 'http://www.expedia.com']
  ```

- `?URL[properties(Options)->?Props]@\http` — Same as before, but takes a list of options as a parameter. The options are the same as for `http(...)` but some of them (e.g., put, post, delete) are ignored. Example:

  ```
  ?- 'http://google.com'[properties([redirect=false])->?R]@\http.
  ?R = [-1, -1, 'http://google.com']
  ?- 'http://google.com'[properties([redirect=true])->?R]@\http.
  ?R = [-1, -1, 'http://www.google.com']
  ```

- `?URL[encoding->?Enc]@\http` — sometimes it is necessary to mangle URLs by replacing special characters like '/', ':', etc., so they could be used in various situations, like being sent over the network. This process is known as URL-encoding. The above method returns a list consisting of three components: `[`*Dir*,*File*,*Ext*`]`. *Dir* is the directory portion of the URL in the URL-encoded form, *File* is the file portion (sans the extension; also URL-encoded), *Ext* is the file extension portion.

  Network is not contacted in order to produce these results, so this method is very fast.

- `?Item[base64encode->?Item2]@\http` — base 64 encoding. When sending information over the network, it is necessary to convert some of the special characters into "benign" ACSII characters that would not be mangled by the network. This is called *base 64 encoding*. This is similar to URL encoding, but is not specific to URLs. Here *Item* (the source) can be a character list, an atom, and in the Web situation it is most commonly a file specified as `file(`*Path*`)`. The output, *Item2*, is always an atom or a variable that will be bound to an atom. This is used, for example, to upload files to Web services. Note that if the source contains the ASCII character '\0' then this source cannot be represented as an atom, for otherwise it will be encoded only partially. So, a list-representation or a file should be used.

- `?Item[base64decode->?Item2]@\http` — base 64 decoding. This is the opposite process, used when receiving information from the networks and decoding it. Here, *Item*—the source—is always an atom, but *Item2* can be an atom, a variable (that will be bound to an atom), `file(`*Path*`)`, or `list(`*CharlistOrVariable*`)`. In case of `file(`*Path*`)`, the result of decoding is stored directly to the specified file. The last representation is used when the result of the decoding cannot be represented as an atom because the decoded string contains the ASCII character '\0' and one does not want to store the result in a file. So, when *Item2* has the form `list(...)`, it directs the system to decode the source as a list of characters.

Chapter 5

Querying SQL Databases

by Michael Kifer

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

5.1 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.`

5.2 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`, which is the internal representation of the ERGO null quasi-constant `\@?`. 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 6

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.

6.1 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.
```

6.2 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 ERGO 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 ERGO 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 ERGO 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 ERGO 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 ERGO variables, like `?Subj`, with SPARQL variables, like `?r` and `?y`, in the above query. From the ERGO 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, ERGO 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 ERGO 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, ERGO 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 ERGO 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 ErgoAI Tutorial, in the section on $\mathcal{E}$RGO connectors, at `https://sites.google.com/a/coherentknowledge.com/ergo-suite-tutorial/home/ergo-connectors`.

6.3 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.

6.3.1 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!

6.3.2 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 7

Loading RDF and OWL files

by Paul Fodor and Michael Kifer

This chapter describes the $\mathcal{E}$RGO import facility for RDF and OWL files. The Resource Description Framework (RDF) and the Web Ontology Language (OWL) are families of knowledge representation languages for authoring ontologies.

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`.

7.1 Loading RDF and OWL Files

The main predicate for importing and loading RDF and OWL files into $\mathcal{E}$RGO is `rdf_load`:

`System[rdf_load(?InputFileName,?InputLangSyntax,?IriPrefixes,?RdfModule)]@\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 an $\mathcal{E}$RGO module indicated by the last argument.

`?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 the delimiter for specifying path names. Backslash also works, but it should be doubled, as backslashes need to be escaped.

Note: the input file name can be a URL in which case it should have the form `url(`*the-web-address*`)`. For example, `url('http://www.w3.org/TR/owl-guide/wine.rdf')`. This feature will work only if the host system has all the required software installed (like the *curl* package. (Please refer to the installation instructions.)

`?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', 'N-QUADS', 'NT', 'N3', or 'RDF/JSON' (lowercase versions are also accepted).

If `?InputLangSyntax` is an empty atom `''` then the input syntax is determined from the file extension.[1]

`?IriPrefixes` must be bound to an ERGO 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.

The last argument, `?RdfModule`, must be bound to an ERGO symbol (Prolog atom); it indicates the ERGO module into which the RDF imported triples should placed at run time. These triples have the form `?Subject[?Property->?Object]` and can be queried as follows:

```
?Subject[?Property->?Object]@MyRdfModule
```

where we assume that `?RdfModule` is bound to `MyRdfModule` in this example.

A 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 query would suffice:

`System[rdf_load(?InputFileName, ?RdfModule)]@\owl`

The input language syntax is determined from the file extension and no IRI prefixes are expected to be supplied. In other words, a call like

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

is equivalent to

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

7.2 Other API Calls

Besides loading, the following API calls are supported:

- `?RdfModule[rdf_insert(?S,?P,?O)]@\owl` — insert `?S[?P->?O]` into the RDF module indicated by `?RdfModule`.

[1] `.owl` and `.rdf` for 'RDF/XML', `.nt` for 'N-TRIPLES' and 'NT', `.ttl` for 'TTL' and 'TURTLE', `.nq` for 'N-QUADS', `.jsonld` for 'JSON-LD', `.rj` for 'RDF/JSON', `.n3` for 'N3'.

- `?RdfModule[rdf_delete(?S,?P,?O)]@\owl` — delete a fact that matches `?S[?P->?O]` from the RDF module indicated by `?RdfModule`.
- `?RdfModule[rdf_deleteall]@\owl` — empty out the specified RDF module.
- `?Subject[rdf_reachable(?RdfModule,?Property)->?Object]@\owl` — true if `?Object` is reachable from `?Subject` via a path of properties `?Property`. If the property is specified simply as `?` then any path will do. If `?Property` is bound (say, to `foo`) then only the paths consisting of the `foo`-edges will be considered. If `?Property` is an unbound variable then any path will do, if all edges of that path are the same.
- `?RdfModule[rdf_predicate->?P]@\owl`, `?RdfModule[rdf_subject->?S]@\owl`, `?RdfModule[rdf_object->?O]@\owl` — return the set of all properties, subjects, and objects, respectively, in the RDF module `?RdfModule`.

7.3 Importing Multiple RDF/OWL Files

Multiple RDF and OWL files can be loaded into separate $\mathcal{E}$RGO modules (and the same file can even be loaded into different modules, if so desired). However, what happens if two files are loaded into the *same* module? For instance,

```
?- System[rdf_load('wine.owl', MyRdfModule)]@\owl,
   System[rdf_load('beer.owl', MyRdfModule)]@\owl,
   ... ... ... do something ... ... ...
```

In that case, the data from the second import will be *added* to the data obtained from the second import. If this additive behavior is not what is required in a particular situation and one wants the second import to *override* the first, a call to `rdf_deleteall` will do the trick:

```
?- System[rdf_load('wine.owl', MyRdfModule)]@\owl},
   ... ... ... do something ... ... ...
   MyRdfModule[rdf_deleteall]@\owl,  // erase the previously imported data
   System[rdf_load('beer.owl', MyRdfModule)]@\owl},
   ... ... ... do something else ... ... ...
```

Chapter 8

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

by Theresa Swift

Evidential probability [2] 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 [2], 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 [2] 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 9

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.

9.1 API for Loading and Saving Tabular Data

First, the DSV package (`e2dsv`) must be loaded into an $\mathcal{E}$RGO module, say `dsv`. One can choose a different name, say `mydata`, but then `@dsv` in the examples below would need to be changed to `@mydata`.

```
?- [e2dsv>>dsv].     // load the e2dsv package
```

After that, the following predicates become available in the module `dsv`:

- `dsv_load`(*?Infile,?Spec,?Format*)`@dsv`: 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*)`@dsv`: 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.

In the above, *?Infile* is either an atom representing a local file name or has the form `url`(*WebDocAddress*), where *WebDocAddress* must be an atom (e.g., `url('http:foo.com/bar')`). The Web page should *not* be protected by passwords or SSL. Also, CSV/DSV files produced in the old format of Mac Classic are not supported.

The import specification, *?Spec*, 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 null values `\@?`. All values are imported as general ERGO 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 items in the imported spreadsheet are numbers, dates, currencies, etc., then this is not very satisfactory. The possible values for an *ArgSpec* are:
  - `atom` or `?`: the corresponding items from the DSV file are converted into Prolog atoms.
  - `integer`: the corresponding spreadsheet items are converted into integers. If an item cannot be converted into an integer, a warning is issued and the item is converted into an atom. The warning does not abort the computation and is only intended to alert the user.
  - `float`: the corresponding items from the DSV file are converted into floating point/decimal numbers. If an item cannot be converted into a floating point number, a warning is issued and the value is converted into an atom. Again, the warning is intended to merely alert the user.
  - `string`: the corresponding items from the DSV file are converted into lists of characters.
  - `date`: the corresponding items in the DSV file are expected to be in the canonical lexical form for `\date` literals in ERGO (i.e., `YYY-MM-DD`; e.g., `2017-11-26`). They are then converted into proper `\date` constants (i.e., into `"2017-11-26"^^\date`).
  - `time`: the corresponding items from the spreadsheet are expected to be in the canonical lexical form for a `\time` literal (`HH:MM:SS` or `HH:MM:SS.XXXX`). They are converted to proper ERGO `\time` constants.
  - `dateTime`: the corresponding items from the spreadsheet are expected to be in the canonical lexical form for a `\dateTime` literal (e.g., `2017-12-03T09:23:01`). They are converted to proper ERGO `\dateTime` constants. See the section *Primitive Data Types* in the ERGO Reasoner's User Manual for the details of the `\dateTime` type.
  - `duration`: the corresponding items from the DSV file are expected to be in the canonical lexical form for a `\duration` literal (e.g., `P22Y2M10DT1H2M3S`). They are converted to proper ERGO `\duration` constants. See the section *Primitive Data Types* in the ERGO Reasoner's User Manual for the details of the `\duration` type.

  - `currency`: the corresponding items from the DSV file are expected to be in the canonical lexical form for a `\currency` literal (e.g., `USD 20,666.55`). These items are converted to proper $\mathcal{E}$RGO `\currency` constants. See the section *Primitive Data Types* in the $\mathcal{E}$RGO Reasoner's User Manual for the details of the `\currency` type.
  - `currency`(*Unit*): where *Unit* is a currency unit like USD, GBP, EUR, etc. In this case, the corresponding items in the spreadsheet are assumed to be numbers (integer, decimal, float), which are then converted into $\mathcal{E}$RGO `\currency` constants with *Unit* as the currency unit.
  - `term`: the corresponding items in the spreadsheet are expected to have the form of a canonical Prolog term. They are converted to Prolog terms. If an item does not parse as a term, an atom is returned with a warning.
  - `hilog`: the items must have the form of a canonical Prolog term, and they are converted to the corresponding HiLog terms. If an item does not parse as a term, an atom is returned with a warning.

  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
- `...+titles` — ignore the first line in file (assumed to be the column header). The ... part here must be `csv`, `psv`, or `tsv`.
- `...+titles(N)` — like `...+titles` except that $N \geq 0$ first lines in the tabular file will be ignored.
- `...+pad(N)` — pad each line in the input file with N variables (if $N > 0$) or cut N columns from each line (if $N < 0$). The ... part here stands for any of the earlier listed combinations.
- `...+error` — normally, if problems are encountered (such as the inability to convert to integer or float, wrong line length, etc.), a warning is issued. This option forces errors instead of warnings. Once an error is thrown, the loading stops. As before, the ... part represents any of the earlier listed combinations.
- or it can be a list of options, each having one of these forms:

- `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.
- `titles` or `titles(N)` — tells to skip the first line—or $N >= 0$ lines—in the input file, which are assumed to be the header that contains column names or some other non-tabular information.
- `pad(N)` — pad each line with N variables (if $N > 0$) or cut N columns from each line (if $N < 0$). If line length does not match the number of arguments specified in the `Format` argument then an error or warning is issued. This option helps avoid this.
- `error` — if conversion problems, wrong line length, or similar issues are found, warnings are given. With the `error` option, a stricter policy is assumed and errors are issued instead. If an error is issued, loading of the data stops immediately.

Here is an example of a use of a complex options list, where `salary.dsv` is assumed to be a spreadsheet in which fields are separated with the pair of characters \>:

```
?- dsv_load('salary.dsv',q,[separator="\\>"^^\charlist, titles(3)])@dsv.
```

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

- The predicate must be queried using the idiom *predname*(...)@*module*, where *module* is the module into which `e2dsv` was loaded (`dsv` in our example). The number of the arguments must match the specification *Spec*—see above. For instance, as a result of the above `dsv_load` command, the predicate `q/1` will be created, and the imported data should be queried as

```
?- q(?X)@dsv.
```

- **The previous contents of the aforesaid predicate *predname*(...)@*module* will be wiped out when the DSV data is loaded into that predicate.**
- For efficiency, the aforesaid predicate *predname*(...)@*module* is created in a special way as a Prolog predicate, *not* as a HiLog predicate. This implies that a HiLog query (HiLog due to a variable in the predicate position) such as

```
?- ?pred(?X)@dsv.
```

  will *not* bind `?pred` to `q` in our example and the imported tabular data cannot be queried this way: the desired predicate name (i.e., `q` here) must be named explicitly.
- If the aforesaid predicate *predname*(...)@*module* is queried from within a file (e.g., appears in the body of a rule in a file) rather than from the command line in the $\mathcal{E}$RGO shell, it must be declared there as

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

because, as explained above, the predicate in question is special.

For instance, if the DSV file `example.csv` has the form

```
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.

More examples of dealing with spreadsheets can be found in the $\mathcal{E}$RGO Examples Bank (see *Importing tabular data*) `https://sites.google.com/a/coherentknowledge.com/ergo-suite-tutorial/example-bank-of-advance`

9.2 Loading Multiple Spreadsheets into the Same Module

This package allows one to load multiple spreadsheets into the same $\mathcal{E}$RGO module at the same time, but then the predicates into which these spreadsheets are loaded must be different (either the name or the arity must differ). For instance, suppose we have `example1.csv` and `example2.psv`, both containing tables with only two fields. Then

```
?- [e2dsv>>dsv].
?- dsv_load('example1.csv',p/2,csv)@dsv,
   dsv_load('example2.psv',q/2,psv)@dsv.
```

will load the first spreadsheet into the predicate `p(?X,?Y)@dsv` and the second into `q(?X,?Y)@dsv`. Both spreadsheets will be queriable via these respective predicates. In contrast, if the chosen predicates are the same, as in

```
?- dsv_load('example1.csv',p/2,csv)@dsv,
   dsv_load('example2.psv',p/2,psv)@dsv.
```

then `example1.csv` will first be loaded into `p(?X,?Y)@dsv` and then this will be immediately overwritten by the contents of `example2.psv`. As a result, only the data in the second spreadsheet will be accessible through `p(?X,?Y)@dsv`, and nothing will be loaded into `q(?X,?Y)@dsv`.

Note: the command `[e2dsv>>`*datamodule*`]` should be executed only once per target module, i.e., regardless of how many spreadsheets are loaded into *datamodule* (*datamodule*=`dsv` in our examples).

9.3 Accessing Tabular Data via Frames

The previous discussion centered around accessing spreadsheet data via predicates, but this is not always convenient. Tabular data often has rows composed of dozens of columns and useful columns may be interspersed with those of no interest. For instance, if the first two columns are of interest, the next 20 are of no interest, and columns 23, 24 are again of interest, the method discussed so far would force us to access the data via a predicate that has 24 arguments—a serious inconvenience. The frame-based access method solves this problem, as it lets one query the data using the frames of the form

```
?rowId[arg(argNumber) -> ?Value]@dsv
```

For instance, in case we only want to see columns 1, 2, 23, 24, we can ask this query:

```
?- ?[arg(1)->?V1, arg(2)->?V2, arg(23)->?V23, arg(24)->?V24]@dsv.
```

So long as one is dealing with just one spreadsheet in the data module `\dsv`, the form of the object Id of the frame is of no importance, so in the above example we simply used the don't care variable `?`. If, however, one loads more than one spreadsheet into the same module, it often becomes necessary to query a particular spreadsheet instead of all of them at once. When access to the tabular data is done via predicates, this is not a issue—one simply uses the desired predicate to focus the query to that predicate. In case of the frame access, this is done by restricting the form of the object Id of the frame. The general form of such an object Id is

`\e2dsv(`*PredName*`,`*PredArity*`,`*RowId*`)`

What is `RowId`? The `RowId` field used in the above object Id specification is used to make sure the OIDs of different rows in spreadsheets imported into $\mathcal{E}$RGO are distinct. It can be generated automatically or a user-supplied primary key can be used.

By default, *RowId* is generated automatically by $\mathcal{E}$RGO; it is *not* the same as the line number in a spreadsheet. The user does not usually deal with that component. Alternatively, the user can specify *primary key* information by calling the predicate

`?- add_key_info(`*PredName*`,`*PredArity*`,`*KeyPositionsList*`)@dsv.`

For instance, if we are importing data into a 30-ary predicate `foo` that has a natural primary key consisting for a pair of arguments in positions 2 and 4 then calling `add_key_info(foo,30,[2,4])@dsv` (in advance) will cause *RowId* to look like this: `(`*2nd-arg-in-foo*,*4th-arg-in-foo*`)`; the overall object Id will then look like this:

`\e2dsv(foo,30,(`*2nd-arg-in-foo*,*4th-arg-in-foo*`))`   `//`*note*: (...) *around key columns*

When primary keys are used, the user is likely to deal with *RowId*s directly, as these would be meaningful values.

The predicate name and the arity in the above object Ids are there in order to distinguish objects that came from different tables/spreadsheets. They can be used to focus queries so that they would retrieve the rows of a particular spreadsheet. Here *PredName* and *PredArity* are the same as what was used as part of the *?Spec* argument to `dsv_load` on page 53. For instance, suppose that two sheets were loaded into the module `dsv`: one into the predicate `foo/30` and the other into `bar/55` (i.e., `foo` has 30 arguments and `bar` 55), and suppose we want to use frames to query `foo/30` and to extract columns 1, 2, 23, and 24. One can do this using the following frame query:

```
?- \e2dsv(foo,30,?)[arg(1)->?V1,arg(2)->?V2,arg(23)->?V23,arg(24)->?V24]@dsv.
```

Note that we used `?` in the *RowId* argument of the above object Id expression, as we assume here that the user does not want this information. However, one can also put a normal variable there, if desired.

9.3.1 Accessing via Frames and Meta Data

The frame-based interface to tabular data that was just described requires one to use column numbers, and this is often inconvenient and error-prone. In many cases, tabular data comes with each column having a name, and it might be desirable to be able to access the data via the column names rather than numbers. Fortunately, this is easy to achieve as follows.

1. Suppose we are importing tabular data that has three columns, where the first is named `Id`, the second `Name`, and the third `Age`. Create a file, say `metainfo.ergo`, with information like this:

   ```
   :- export{column_name(?,?)}.
   column_name(Id,1).
   column_name(Name,2).
   column_name(Age,3).
   ```

   The reason for the export statement is that the next step adds this information to the module `dsv` into which we previously loaded the package `e2dsv`. That package is encapsulated and only the explicitly exported predicates and frames can be accessed from other modules. We will see in Step 3 that indeed `column_name` is accessed from another module (`main`), and this is why it needs to be exported.

2. *Add* this information to the module that contains the imported tabular data (`dsv` in our example):

   ```
   ?- [+metainfo>>dsv].   // or add{metainfo>>dsv}.
   ```

3. Insert the *bridge rule*

   ```
   ?rowId[?property->?val] :-
               (
                 column_name(?property,?colNum),
                 ?rowId[arg(?colNum)->?val]
               )@dsv.
   ```

   into the module(s) from where you intend to access the tabular data.

For instance, if one inserts the above bridge rule into module `main` then, in that module, one can ask queries such as

```
?- ?[Id->?i,Age->?a].
```

Chapter 10

Importing and Exporting 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`.

10.1 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.

10.2 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!) terms of the form `true()`, `false()`, and `'NULL'(?)` (the latter is the internal representation of the $\mathcal{E}$RGO quasi-constant `\@?`).
- Finally, an object of the form `{` str_1`:`val_1`,...,`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 HiLog $\mathcal{E}$RGO terms as follows:

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

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

ERGO provides the following methods for importing JSON:

- `Source[parse -> ?Result]@\json`
  Here *Source* can have one of these forms

  - `string(Atom)`
  - `str(Atom)`
  - `url(Atom)`
  - `file(Atom)`
  - *Atom*
  - 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 ERGO representation of that structure. The form `url(Atom)` can be used to ask ERGO to get a JSON document from the Web. In that case, *Atom* must be a URL. 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 will come from 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(`*?Var*`)` then *?Var* will get bound to 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 ERGO term as explained above. For instance, the query `string('{"abc":1, "cde":2}')[parse->?X]@\json` will bind `?X` to the ERGO HiLog term `json([abc=1,cde=2])` 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

[1] Sending the end-of-file signal is not possible in the ERGO Studio Listener, so this last option is not available through the studio.

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])`.

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

```
{ "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&ke
            [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])]
```

The same can actually be obtained in a much simpler way using the `url` feature for the JSON source, as described above:

```
?- url('https://kgsearch.googleapis.com/v1/entities:search?query=benjamin_grosof&key=AIzaSyAa
        parse(itemListElement) -> ?Answer
   @\json.
```

However, at present the `url(...)` feature works only for documents that are not protected by passwords or SSL.

10.3 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 of this translation 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 problem is side-stepped by converting lists with embedded frames, such as above, into plain lists augmented with additional frame-facts. For instance, the above offending list is represented as

```
[newObjId1, newObjId2]                  // complex list became plain list
```

where `newObjId1` and `newObjId2` are newly invented constants that do not appear anywhere else. In addition, the following facts are added:

```
newObjId1[place->Boston,numberOfYears->5]. // these are the facts that 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`.
- `?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.

Only JSON objects (i.e., {...} - structures, not standalone constants or lists) can be converted to facts.

Conversion to facts involves creation of a root $\mathcal{E}$RGO object that represents the entire JSON structure. This object then points to other objects that represent the various components of that structure, etc. The Id of the root object can be obtained via the query

```
?- ?Module[json_root->?Oid]@\json.
```

Here `?Module` is the module into which the JSON object is dumped by the`parse2memory` method or into which the file of facts is loaded when `parse2file` is used. If more than one JSON object is dumped into the same module, the above query will return multiple answers—one for each JSON structure dumped into the module. One can "forget" the root-level oids using this API call:

```
?- ?Module[forget_roots]@\json.
```

This is useful in situations when one is done processing a previous JSON structure and needs to traverse a newly-dumped structure into the same module. However, the most common way of working with JSON is when modules hold just one JSON structure at a time, and `erasemodule{...}` is used before another JSON structure is dumped into the same module.

10.4 Exporting to JSON

ERGO provides API for exporting HiLog terms as well as objects to JSON.

10.4.1 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,

```
ergo> p(o)(${a(9)},b,?L,[pp(ii),2,3,?L])[term2json -> ?J]@\json.
?J = '{"functor":{"functor":"p","arguments":["o"]}
       "arguments":[{"predicate":"a","module":"main","arguments":[9]},
                    "b",
                    {"variable":"h0"},
                    [{"functor":"pp","arguments":["ii"]},
                    2,
                    3,
                    {"variable":"h0"}]]}'

ergo> foo(a,b,bar(c,d))[term2json->?J]@\json.
?J = '{"functor":"foo",
       "arguments":["a","b",{"functor":"bar","arguments":["c","d"]}]}'

ergo> [a,b,bar(c,d)][term2json->?J]@\json.
?J = '["a","b",{"functor":"bar","arguments":["c","d"]}]'

ergo> (a,b,bar(c,d))[term2json->?J]@\json.
?J = '{"commalist":"["a","b",{"functor":"bar","arguments":["c","d"]}]"}'
```

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"`. Frame and subclass/isa formulas are also supported, but not aggregate functions. To see whether a particular form of a reified formula is supported and how it is represented in JSON, use the JSON API method `term2json`, as shown above. The general form of that method is given below:

- `?Term[term2json -> ?Json]@\json` — convert HiLog term `?Term` 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.

10.4.2 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 11

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.

11.1 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 ERGO will use its in-memory cache instead. However, insertion and deletion of facts in such a module will still cause database operations.

11.2 Examples

Consider the following scenario sequence of operations.

```
// Create new modules mod, db_mod1, db_mod2.
ergo>  newmodule{mod}, newmodule{db_mod1}, newmodule{db_mod2}.
ergo>  [persistentmodules>>pm].

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

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

Restart the ERGO system.

```
// Create the same modules again
ergo>  newmodule{mod}, newmodule{db_mod1}, newmodule{db_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 12

SGML and XML Import for ℰRGO

by Rohan Shirwaikar and Michael Kifer

This chapter documents the ℰRGO package that provides XML and XPath parsing capabilities. The main predicates support parsing SGML, XML, and HTML documents, and create ℰRGO objects in the user specified module. Other predicates evaluate XPath queries on XML documents and create ℰRGO objects in user specified modules. The predicates make use of the **sgml** and **xpath** packages of XSB.

12.1 Introduction

This package supports parsing SGML, XML, and HTML documents, converting them to sets of ℰRGO objects stored in user-specified ℰRGO 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 ℰRGO objects:

```
obj1[ greeting -> obj2]
obj2[ attribute(id) -> '1']
obj2[ first -> obj3]
obj3[ attribute(ssn) -> '111']
obj3[ \text -> 'John']
```

To load the `XML` package, just call any of the API calls at the ℰRGO prompt.

The following calls are provided by the package. They take SGML, XML, HTML, or XHTML documents and create the corresponding ERGO objects as specified in Section 12.3.

`?InDoc[load_sgml(?Module) -> ?Warn]@\xml`
Import XML data as ERGO objects.

`?InDoc[load_xml(?Module) -> ?Warn]@\xml`
Import SGML data as ERGO objects.

`?InDoc[load_html(?Module) -> ?Warn]@\xml`
Import HTML data as ERGO objects.

`?InDoc[load_xhtml(?Module) -> ?Warn]@\xml`
Import XHTML as ERGO objects.

The arguments to these predicates have the following meaning:

`?InDoc` is an input SGML, XML, HTML, or XHTML document. It must have one of these forms: `url('`*url*`')`, `file('`*file name*`')` or `string('`*document as a string*`')`. If `?InDoc` is just a plain Prolog atom (ERGO symbol) then `file(?Source)` is assumed. `?Module` is the name of the ERGO module where the objects created by the above calls should be placed; it must be bound. `?Warn` gets bound to a list of warnings, if any are generated, or to an empty list; it is an output variable.

12.2 Import Modes for XML in Ergo

XML can be imported into ERGO in several different ways, which can be specified via the `set_mode(...)@\xml` primitive. These modes control two aspects of the import:

- white space handling, and
- navigation links that may be added to the imported data.

12.2.1 White Space Handling

The XML standard requires that white space (blanks, tabs, newlines, etc.) must be preserved by XML parsers. However, in the applications where ERGO is used, XML typically is viewed as a format for data in which white space is immaterial. For that reason, by default, the ERGO's XML parser operates in the *data mode* in which every string is trimmed on both sides to remove the white space. In addition, the empty strings `''` are ignored. This implies that, for example, there will be no `\text` attribute to represent a situation like this:

```
<doc>
    <spaceonly>   </spaceonly>
</doc>
```

and the only data created to represent the above document will be

```
obj1[doc->obj2]@bar
obj2[spaceonly->obj3]@bar
```

(plus some additional navigational data about order, siblings, parents, etc.). This means that, if capturing certain white space is needed, it should be encoded explicitly in some way, e.g.,

```
<spaceonly>___</spaceonly>
```

instead of three spaces.

Alternatively, one can request to change the XML parsing mode to *raw*:

```
   ?- set_mode(raw)@\xml.
```

In this case, the parser will switch to the pedantic way XML parsers are supposed to interpret XML and all white space will be preserved. However, beware what you wish because even for the above tiny example the representation will end up not pretty because every little bit of white space will be there (even the one that comes from line breaks):

```
obj1[doc->obj2]@bar
obj2[\text->obj3]@bar
obj2[\text->obj6]@bar
obj2[spaceonly->obj4]@bar
obj3[\string->'
  ']@bar
obj4[\text->obj5]@bar
obj5[\string->'   ']@bar
obj6[\string->'
']@bar
```

It is more than likely an ℰRGO user will not want objects like `obj3` and `obj6`.

Finally, if the *raw* mode is not what is desired, one can always switch back to the data mode:

```
   ?- set_mode(data)@\xml.
```

12.2.2 Requesting Navigation Links

This aspect can be changed via the calls

```
   ?- set_mode(nonavlinks)@\xml.  // the default
   ?- set_mode(navlinks)@\xml.
```

where `nonavlinks` is the default.

The `nonavlinks` method uses a slightly simpler translation from XML to ℰRGO objects and no extra navigation links are provided. This mode is used when the imported XML document has known tructure and is viewed simply as set of data to be ingested (e.g., payroll data).

In the `navlinks` mode, the representation is slightly more complex but, most importantly, that imported data includes additional information that provides parent/child/sibling links among XML objects as well as the ordering information, which allows one to reconstruct the original XML document. This mode is used when the structure of the input XML has high variability or may even be arbitrary. This arises, for instance, when one needs to transform arbitrary XML import or to extract certain information from unknown structures. The exact representation of this navigational information is described in subsequent sections.

12.3 Mapping XML to $\mathcal{E}$RGO Objects

This mapping is based on an XML-to-$\mathcal{F}$LORA-*2* object correspondence developed by Guizhen Yang. It specifies how an XML parser can construct the corresponding F-logic objects after parsing an input XML document. The basic ideas are as follows:

- XML elements, attribute values, and text strings are modeled as objects in F-logic.
- XML elements are reachable from parent objects via F-logic frame attributes of the same name as the XML element name.
- XML element attributes are also modeled as F-logic frame attributes but their name is `attribute(`*XML attribute name*`)`.

This mapping does not address comments or processing instructions—they are simply ignored. However, this mapping does address the issue of mixed text/element content in which plain text and subelements are interspersed. This mapping also assumes that XML entities are resolved by the XML parser.

12.3.1 Invention of Object Ids for XML Elements

According to the XML specification 1.0, an XML element can be identified by an oid that is unique across the document. The import mechanism invents such an oid automatically. 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 corresponding to the XML document. The oids of leaf nodes, which have no outgoing arcs and carry plain text only, are just the string values themselves.

For example, the following XML document

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

is represented via the following F-logic objects:

```
obj1[person -> obj2].
obj2[attribute(ssn) -> '111-22-3333', name -> obj3].
obj3[attribute(first) -> John, attribute(last) -> Smith].
```

Here `obj1` is the root object, `obj2` is the object corresponding to the `person` element, and `obj3` is the object that represents the `name` element. The strings `'111-22-3333'`, `John`, and `Smith` are oids that stand for themselves.

12.3.2 Text and Mixed Element Content

The content of an XML element may consist of plain text, or subelements interspersed with plain text as in

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

How text is actually handled in the translation to F-logic depends on the mode of import: `nonavlinks` or `navlinks`. The former is simpler because it discards all the information about the order of the text nodes with respect to subelements and other text nodes.

- **In the `nonavlinks` mode**:
  Each text segment is modeled as a value of the attribute `\text` of the parent element-object of that text segment.[1] Thus, for the above XML fragment, the translation would be

  ```
  obj1[greeting -> obj2].
  obj2[\text -> {'Hi! My name is ', '.'},
       first -> obj3,
       last  -> obj4
  ].
  obj3[\text -> John].
  obj4[\text -> Smith].
  ```

- **In the `navlinks` mode**:
  Here the order of the text and subelement nodes must be preserved and so each text node is modeled as if it were a value of a special attribute `\string` in an empty XML element named \text, e.g.,

  ```
  <\text \string="John"/>
  ```

  As a consequence, a separate F-logic object is created to represent each text segment. (Compare this to the translation in the `nonavlinks` mode, which does not create separate objects for text nodes.) Thus, for the aforesaid `greetings` element the translation will be

[1] Of course, XML does not allow such names for tags and attributes, and this is the whole point: adding such an invented name to the F-logic translation will not clash with other tag names that might be used in the XML documents.

```
obj1[greeting -> obj2].
obj2[\text -> {obj3, obj8},
     first -> obj4,
     last  -> obj6
].
obj3[\string -> 'Hi! My name is '].
obj4[\text -> obj5].
obj5[\string -> John].
obj6[\text -> obj7].
obj7[\string -> Smith].
obj8[\string -> '.'].
```

How exactly the aforesaid order is preserved in the `navlinks` mode is explained later.

12.3.3 Translation of XML Attributes

An XML attribute, *attr*, in an element is translated as an attribute by the name `attribute`(*attr*) attached to the object that corresponds to that element.

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. 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:

```
obj1[paper -> obj2]
obj2[title -> obj4]
obj2[attribute(id) -> yk00]
obj2[attribute(references) -> 'klw95 ckw91'
obj4[\text -> obj5]      // here we assume that the navlinks mode was used
obj5[\string -> 'paper title']
```

However: if the document has an associated DTD *and* the attribute `references` were specified there as `IDREFS` as in

```
<!ATTLIST paper references IDREFS #IMPLIED>
```

then that attribute is translated as

```
obj2[attribute(references)->[klw95,ckw91]]
```

i.e., the value becomes a list.

With this, we are done describing the **nonavlinks** mode. The remaining subsections in the current section apply to the **navlinks** mode only.

12.3.4 Ordering

This section applies to the **navlinks** mode only.

XML is order-sensitive and the order in which elements and text appear is significant, in general. The order of the attributes within the same element tag is *not* significant, however.

While the **nonavlinks** mode is sufficient for most data-intensive uses of XML in ƐRGO, more complex tasks may require the knowledge of how items are ordered within XML documents. Specifying a total order among the elements and text in an XML document suffices for that purpose, if this order agrees with the local order within each element's content.

Consider the following XML document

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

It can be represented by the tree in Figure 12.1 in which the parenthesized integers show the total order assigned to the F-logic objects.

The ordering information that exists in XML documents is captured in F-logic via a special attribute called `\order`, which tells position within the total ordering for each element and text node. It is for that purpose that text segments are modeled in the **navlinks** mode as element-style objects (each segment having its own oid) and not simply as attributes, as is the case with the simpler **nonavlinks** mode.

12.3.5 Additional Attributes and Methods in the navlinks Mode

Since the **navlinks** mode is intended for applications that need to navigate from children to parents, to siblings, and more, the importer adds the following additional attributes and methods to the F-logic objects into which XML elements and text are mapped.

1. `\in_arc`
   For each node, `\in_arc` returns the unordered set of labels of the arcs pointing to this node, i.e., this node's in-arcs. Roughly, `\in_arc` is defined as follows:

   ```
   ?O[\in_arc -> ?InArc] :- ?[?InArc -> ?O].
   ```

```
                    obj0 (0)
                    |
                    | person
                    |
        id /----  obj1 (1)
          /         ^
         /         / \
111-22-3333  name /   \ email
                 /     \
          (2) obj2     obj7 (7)
               / \        \
        first /   \last    \ \text
             /     \        \
      (3) obj3     obj5 (5)  obj8 (8)
            |      |          \
      \text |      | \text     \  \string
            |      |            \
      (4) obj4     obj6 (6)    'jsmith@abc.com'
            |      |
    \string |      | \string
            |      |
           John  Smith
```

Figure 12.1: Total ordering of the F-logic objects arising from XML ordering

Note that for a node representing a text segment, the value of its `\in_arc` 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 the nodes without a right sibling.

5. `\childcount`
   For each element node, `\childcount` returns the number of the immediate children of that element, which includes subelements and text segments.

6. `\childlist`
   For each element node, `\childlist` returns a list of the oids of the immediate children (subelements and text segments) of that element.

7. `\child(N)`

For each node, \child(N) returns the N-th child, where 0 ≤ N < \childcount. Note: the first child is the 0-th child.

8. \in_child_arc(N)
   For each node, \in_child_arc(N) returns the in-arcs of the N-th child, where 0 ≤ N < \childcount. This attribute is defined as follows:

   ```
   ?O[\in_child_arc(?N)->?InArc] :- ?O[\child(?N)->?[\in_arc->?InArc]].
   ```

12.4 Inspection Predicates

This section applies both to the nonavlinks mode and the navlinks mode.

It is sometimes hard to see which objects have actually been created to represent an XML document or an element. This is especially true in case of navlinks mode, which includes a host of special navigational attributes. The purpose of inspection predicates is to provide a simple way to view the objects, and they also filter the navigational attributes out. Consider the document foo.xml below:

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

Even for such a simple document, the query

```
?- 'foo.xml'[load_xml(bar) -> ?W]@\xml.    // load foo.xml into module bar
?- ?_X[?_Y->?_Z]@bar, ?Z = ${?_X[?_Y->?_Z]}.  // get all facts
```

that asks for all the facts—stored and derived—will yield 56 results in the navlinks mode, which is overwhelming to inspect visually. However, the core facts that describe these objects are only 8, and they can be obtained by asking the query

```
?- bar[show->?P]@\xml.
```

One furthermore might want to see the representation of individual elements (e.g., element named first):

```
?- bar[show(first)->?P]@\xml.
```

and this is much more manageable:

```
?P = ${obj4[\text->obj5]@bar}
?P = ${obj4[attribute(ssn)->'111']@bar}
```

or of elements that have particular attributes (ssh in this example):

```
?- bar[show(attribute(ssn))->?P]@\xml.
```

which yields the same result as above (because the element first has the attribute ssn).

12.5 XPath Support

The 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.

Note: XPath support does not currently work under Windows 64 bit (but does under 32 bits) due to the fact that we could not produce a working `libxml2.lib` file (`xmlsoft.org` provides `linxml2.dll` for Windows 64, but not `libxml2.lib`).

The following predicates are provided. They select parts of the input document using the provided XPath expression and create ℰRGO objects as specified in Section 12.3. These predicates handle XML, SGML, HTML, and XHTML, respectively.

`?InDoc[xpath_xml(?XPathExp,?NS,?Mod)->?Warn]`	apply XPath expression to an XML document and import the result
`?InDoc[xpath_xhtml(?XPathExp,?NS,?Mod)->?Warn]`	apply XPath expression to XHTML and import the result

The arguments have the following meaning:

`InDoc` specifies the input document; this parameter has the same format as in Section 12.1. `?XPath` is an XPath expression specified as a Prolog atom. `?Module` is the module where the resulting ℰRGO 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, if none.

`?NamespacePrefList` is a string that has the form of a space separated list of items of the form *`prefix = namespaceURL`*. This allows one to use namespace prefixes 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 `?NamespacePrefList`:

```
url('http://w3.org')[xpath_xhtml('/x:html/x:head/x:meta',
                                 'x=http://www.w3.org/1999/xhtml',
                                 foomodule)
                     -> ?Warnings]@\xml.
```

12.6 Low-level Predicates

This section describes low-level predicates in the XML package. These predicates parse the input documents into Prolog terms that then must be further traversed recursively in order to get the desired information.

- `parse_structure(?InDoc,?InType,?Warnings,?ParsedDoc)@\xml` — take the document `?InDoc` or type `?InType` (`xml`, `xhtml`, `html`, `sgml`) and parse it as a Prolog term (will not be imported into any module as an object).

- `apply_xpath(?InDoc,?InType,?XPathExp,?NamespacePrefList,?Warnings,?ParsedDoc)@\xml` — like the above, but first applies the XPath expression `?XPathExp` to `?InDoc`. The `?InType` parameter must be bound to `xml` or `xhtml`.

The output, `?ParsedDoc`, is a Prolog term that represents the parse of the input XML document in case of `parse_structure` and the result of application of `?XPathExp` to the input document in case of `apply_xpath`. The format of that parse is described in the *XSB Manual, Volume 2: Interfaces and Packages*, in the chapter on *SGML/XML/HTML Parsers and XPath*.

Bibliography

[1] Miguel Calejo. Interprolog: Towards a declarative embedding of logic programming in java. In Jóse Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings*, pages 714–717. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

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