SWI-Prolog Jasmine (ODB) Interface

Jan Wielemaker
SWI,
University of Amsterdam
The Netherlands
E-mail: J.Wielemaker@uva.nl

August 15, 2008

Abstract

The Jasmine interface provides a client interface to the Object Oriented Jasmine database. The interface consists of two layers. The first is a simple wrapper around the Jasmine C-interface. The second provides utility predicates providing more high-level access to Jasmine.

Contents

1 Introduction

Jasmine is an object-oriented database (ODB), implementing ODQL (**O**bject **D**atabase **Q**uery **L**anguage). It provides a C-interface based on the following components:

- *Session management*
  Connecting and disconnecting a database.
- *Variables*
  Within the interface, variables can be declared. These variables can be manipulated both using ODQL statement and from the access language. Variables play a vital role in communicating data.
- *Data conversion*
  Variables can be read and written. They are dynamically typed and the interface provides access to their type and value. In Prolog we can exploit dynamic typing of Prolog to hide most of the data conversion from the user.
- *Collection conversion*
  Collections play a vital role in communicating results from databases. Variables are bound to collections using ODQL statements. They can be queried for their size and converted into Prolog lists.

2 Basic ODB predicates

Below is the definition of the basic ODB access predicates defined in `jasmine.c`.

2.1 Session management

Sessions are accessed using a *session-handle*. This opaque handle is normally preserved in the Prolog database.

odb_ses_start(*-SH, +Database, +User, +Passwd, +EnvFile*)
: Connect to the indicated database and return a handle for the created session in *SH*. *SH* is an opaque Prolog term providing context for subsequent ODB calls. *Database* specifies the database to connect to. It is an atom formatted as below, where *nvode* is the name of the machine to connect to. *User* and *Passwd* and *EnvFile* are either atoms or unbound variables. The latter makes the interface choose default values. *EnvFile* is the name of a file providing parameters for the interface. See the C-API documentation for details.

  [*vnode*::]/jasmine/jasmine

odb_ses_end(*+SH*)
: Terminate the session. Note that `at_halt/1` can be used to ensure termination of the session when Prolog halts.

2.2 ODQL Statements

ODQL statement are passed in textual form and specified either as atoms or SWI-Prolog strings. The latter makes it possible to construct statements using `sformat/3`. See also `odb_exec_odql/3`.

odb_exec_odql(*+SH, +Statement***)**
Execute the given ODQL *Statement* on the session *SH*. This predicate either succeeds or raises an exception. See section **??** for details.

2.3 Variables

Variables play a vital role in the interface. Interface variables are defined using ODQL statements. They are scoped to the session, but otherwise global. There are two approaches to deal with this. One is to define a suitable set of variables for the application at startup and the other is to create them as they are needed. In the latter case one should be sure the variable name is currently not in use. In some of the examples we therefore see:

```
undefVar pcount;
Integer pcount;
```

From this example we learn that variables are typed. The type is accessible through the C-interface and used by the access predicate to perform suitable conversion to Prolog.

odb_get_var(*+SH, +Name, -Value***)**
Fetches the value of the named interface variable. Succeeds if the value can be unified successfully, fails if the value is retrieved correctly but unification fails and raises an exception otherwise.

The representation of *Value* depends on the type of *Name* in the database interface.

- *Bool*
  Booleans are represented either using the atom `true` or `false`.
- *ByteSequence*
  Byte-sequences are represented using an atom (as of SWI-Prolog 3.3 atoms can hold 0-bytes are therefore are capable of storing an arbitrary byte-stream).
- *Date*
  Dates are represented in SWI-Prolog as a floating point number representing the time since the start of 1970. See the Prolog reference manual for manipulating dates.
- *Decimal*
  An ODB decimal is a sequence of digits with precision and scale. There is no representation for this in Prolog and therefore we use `decimal`(*Digits, Precision, Scale*). See the Jasmine C-API docs for details.
- *Integer*
  Jasmine integers are, as SWI-Prolog's integers 32 bit signed values and therefore represented naturally.
- *Nil*
  Nil is represented using the Prolog empty list (`[]`).[1]

[1] This could be considered a bug. What would be a better choice?

- *Object*
  Objects are represented using a opaque term.
- *Real*
  Jasmine reals are double-precision floats and therefore naturally represented using SWI-Prolog floats.
- *String*
  Strings are, like `ByteSequences`, represented as Prolog atoms.
- *Tuple*
  Database N-tuples are represented using a term `tuple`(*...Arg...*), where *Arg* is the converted value for the corresponding position in the tuple.

odb_set_var(*+SH, +Name, +Value***)**
Set a variable. In accordance with the guidelines in the interface this first fetches the value to examine the type of the variable. The latter is problematic, as not-yet-filled variables yield the *Nil* type. In this case the type is determined from *Value*.

This translation currently does not deal with the type-ambiguities. It is currently not possible to set nil-variables to a boolean, byte-sequence or date. This problem can be fixed by using an ODQL query to fill the empty variable with an object of the requested type.

2.4 Collections

Database queries normally yield collections as results. The interface simply converts collections into Prolog lists. The current interface does not yet provide mechanisms for fetching part of a collection. Note that, using ODQL statements it is possible to get the length of a collection before conversion:

```
collection_length(SH, Collection, Length) :-
        odb_exec_odql(SH, 'Integer len;'),
        odb_exec_odql(SH, 'len = ~w.count();', [Collection]),
        odb_get_var(SH, len, Length).
```

odb_collection_to_list(*+SH, +Collection, -List***)**
Where *Collection* is the name of a variable containing a collection or the object-identifier of a collection. The elements of the collection are converted using the same rules as `odb_get_var/3`.

3 Utility Predicates

The predicates of the previous section provide all important aspects of the C-API to the Prolog user. The provided access however is very low-level. A very first start has been made to provide a number of utility predicates.

odb_exec_odql(*+SH, +Format, +Args***)**
First constructs a command using `sformat/3` from *Format* and *Args* and then execute it.

odql(*:SH, +Declarations, +Statements***)**

Utility to deal with a sequence of ODQL statements, requiring some variables to execute them. *Declarations* is a list of *VarName*:*Type*. These variables are first unset and then declared using the given type. Please note that this principle is **not re-entrant**. *Statements* is a list containing a mix of ODQL statements, set/get variables, access collections and ordinary Prolog code:

get(*VarName, Value***)**
Fetch the interface variable *VarName* using `odb_get_var/3`.

set(*VarName, Value***)**
Store the interface variable *VarName* using `odb_set_var/3`.

get_list(*Collection, List***)**
Get a variable or object-id into a list of values using `odb_collection_to_list/2`.

{}(*Goal***)**
Call normal Prolog goal in the module from which `odql/3` was called. Note that {`Goal`} is the same as {}(*Goal*).

-(*Format, Args***)**
Execute an ODQL query using `odb_exec_odql/3`.

Command
Execute ODQL command.

Here is an example, extracting the available *class-families* from the Jasmine database:

```
families(SH, List) :-
        odql(SH,
             [ ss:'Bag<String>'
             ],
             [ 'ss = FamilyManager.getAllFamilies();',
               get_list(ss, List)
             ]).
```

4 Error handling

All errors are reported using Prolog exceptions. This package raises two types of exceptions. If Prolog arguments cannot be converted into the desired data, normal Prolog `type_error` and `instantiation_error` exceptions are raised. Jasmine calls returning an error are translated into an error term of the format

`error`(`package`*(jasmine, ErrorId), Context*)

Where *Context* is

`context`(*Message, _*)

In this term, *ErrorId* is the (numerical) error identifier raised by Jasmine and *Message* is Jasmine's textual representation of the error.

5 Issues

The interface defined here provides the foreign-language basis for a more advanced Prolog ODQL interface. Specifying all ODQL as strings and dealing with the interface variables is not a desirable way to deal with ODQL. A more fundamental approach is to define a Prolog API for ODQL and an interface for translating these Prolog queries into textual ODQL calls. For example, the `families/2` example above could be written as:

```
families(SH, Families) :-
        odql(Families:bag(string) = 'FamilyManager'.getAllFamilies).
```

6 Installation

The jasmine package has currently been build only on Windows. As Jasmine is also available on Unix, the standard SWI-Prolog package infra-structure for Unix foreign packages is provided.

6.1 Unix systems

Installation on Unix system uses the commonly found *configure*, *make* and *make install* sequence. SWI-Prolog should be installed before building this package. If SWI-Prolog is not installed as `pl`, the environment variable `PL` must be set to the name of the SWI-Prolog executable. Installation is now accomplished using:

```
% ./configure
% make
% make install
```

This installs the Prolog library files in `$PLBASE/library`, where `$PLBASE` refers to the SWI-Prolog 'home-directory'.

6.2 Windows

Run the file `setup.pl` by double clicking it. This will install the required files into the SWI-Prolog directory and update the library directory.