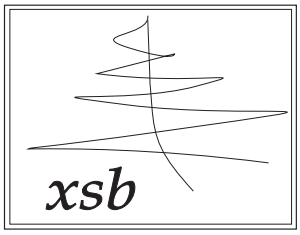
The XSB System
Version 4.0
Volume 2: Interfaces and Packages



July 23, 2021

Credits

Packages and interfaces have become an increasingly important part of XSB. They are an important way to incorporate code from other systems into XSB, and to interface XSB to databases and other stores. Most of the packages had significant contributions by people other than the core XSB developers, for which we are grateful. As a result most chapters have information about its authors.

Contents

Chapter 1

XSB-ODBC Interface

By Baoqiu Cui, Lily Dong, and David S. Warren [1].

1.1 Introduction

The XSB-ODBC interface is subsystem that allows XSB users to access databases through ODBC connections. This is mostly of interest to Microsoft Windows users. The interface allows XSB users to access data in any ODBC compliant database management system (DBMS). Using this uniform interface, information in different DBMS's can be accessed as though it existed as Prolog facts. The XSB-ODBC interface provides users with three levels of interaction: an *SQL level*, a *relation level* and a *view level*. The *SQL level* allows users to write explicit SQL statements to be passed to the interface to retrieve data from a connected database. The *relation level* allows users to declare XSB predicates that connect to individual tables in a connected database, and which when executed support tuple-at-a-time retrieval from the base table. The *view level* allows users to use a complex XSB query, including conjunction, negation and aggregates, to specify a database query. A listing of the features that the XSB-ODBC interface provides is as follows:

- Concurrent access from multiple XSB processes to a single DBMS
- Access from a single XSB process to multiple ODBC DBMS's
- Full data access and cursor transparency including support for
  - Full data recursion through XSB's tabling mechanism (depending on the capabilities of the underlying ODBC driver.

[1] This interface was partly based on the XSB-Oracle Interface by Hassan Davulcu, Ernie Johnson and Terrance Swift.

  - Runtime type checking
  - Automatic handling of NULL values for insertion, deletion and querying

- Full access to data source including
  - Transaction support
  - Cursor reuse for cached SQL statements with bind variables (thereby avoiding re-parsing and re-optimizing).
  - Caching compiler generated SQL statements with bind variables and efficient cursor management for cached statements

- A powerful Prolog / SQL compiler based on [1].
- Full source code availability
- Independence from database schema by the *relation level* interface
- Performance as SQL by employing a *view level*
- No mode specification is required for optimized view compilation

We use the `Hospital` database as our example to illustrate the usage of XSB-ODBC interface in this manual. We assume the basic knowledge of Microsoft ODBC interface and its ODBC administrator throughout the text. Please refer to "Inside Windows^{TM} 95" (or more recent documentation) for information on this topic.

1.2 Using the Interface

The XSB-ODBC module is a module and as such exports the predicates it supports. In order to use any predicate defined below, **it must be imported** from `odbc_call`. For example, before you can use the predicate to open a data source, you must include:

```
:- import odbc_open/3 from odbc_call.
```

1.2.1 Connecting to and Disconnecting from Data Sources

Assuming that the data source to be connected to is available, i.e. it has an entry in `ODBC.INI` file which can be checked by running Microsoft ODBC Administrator, it can be connected to in the following way:

```
| ?- odbc_open(data_source_name, username, passwd).
```

If the connection is successfully made, the predicate invocation will succeed. This step is necessary before anything can be done with the data sources since it gives XSB the opportunity to initialize system resources for the session.

This is an executable predicate, but you may want to put it as a query in a file that declares a database interface and will be loaded.

To close the current session use:

```
| ?- odbc_close.
```

and XSB will give all the resources it allocated for this session back to the system.

If you are connecting to only one data source at a time, the predicates above are sufficient. However, if you want to connect to multiple data sources at the same time, you must use extended versions of the predicates above. When connecting to multiple sources, you must give an atomic name to each source you want to connect to, and use that name whenever referring to that source. The names may be chosen arbitrarily but must be used consistently. The extended versions are:

```
| ?- odbc_open(data_source_name, username, passwd, connectionName).
```

and

```
| ?- odbc_close(connectionName).
```

A list of existing Data Source Names and descriptions can be obtained by backtracking through `odbc_data_sources/2`. For example:

```
| ?- odbc_data_sources(DSN,DSNDescr).

DSN = mycdf
DSNDescr = MySQL driver;

DSN = mywincdf
DSNDescr = TDS driver (Sybase/MS SQL);
```

1.2.2 Accessing Tables in Data Sources Using SQL

There are several ways that can be used to extract information from or modify a table in a data source. The most basic way is to use predicates that pass an SQL statement directly to the ODBC driver. The basic call is:

```
| ?- odbc_sql(BindVals,SQLStmt,ResultRow).
```

where `BindVals` is a list of (ground) values that correspond to the parameter indicators in the SQL statement (the '?'s); `SQLStmt` is an atom containing an SQL statement; and `ResultRow` is a returned list of values constituting a row from the result set returned by the SQL query. Thus for a select SQL statement, this call is nondeterministic, returning each retrieved row in turn.

The `BindVals` list should have a length corresponding to the number of parameters in the query, in particular being the empty list ([]) if `SQLStmt` contains no '?'s. If `SQLStmt` is not a select statement returning a result set, then `ResultRow` will be the empty list, and the call is deterministic. Thus this predicate can be used to do updates, DDL statements, indeed any SQL statement.

`SQLStmt` need not be an atom, but can be a (nested) list of atoms which flattens (and concatenates) to form an SQL statement.

`BindVals` is normally a list of values of primitive Prolog types: atoms, integers, or floats. The values are converted to the types of the corresponding database fields. However, complex Prolog values can also be stored in a database field. If a term of the form `term(VAL)` appears in the `BindVal` list, then `VAL` (a Prolog term) will be written in canonical form (as produced by `write_canonical`) to the corresponding database field (which must be CHAR or BYTE). If a term of the form `string(CODELIST)` appears in `BindVal`, then `CODELIST` must be a list of ascii-codes (as produced by `atom_codes`) and these codes will be converted to a CHAR or BYTE database type.

`ResultRow` for a select statement is normally a list of variables that will nondeterministically be bound to the values of the fields of the tuples returned by the execution of the select statement. The Prolog types of the values returned will be determined by the database types of the corresponding fields. A CHAR or BYTE database type will be returned as a Prolog atom; an INTEGER database field will be returned as a Prolog integer, and similarly for floats. However, the user can request that CHAR and BYTE database fields be returned as something other than an atom. If the term `string(VAR)` appears in `ResultRow`, then the corresponding database field must be CHAR or BYTE, and in this case, the variable `VAR` will be bound to the list of ascii-codes that make up the database field. This allows an XSB programmer to avoid adding an atom to the atom table unnecessarily. If the term `term(VAR)` appears in `ResultRow`, then the corresponding database field value is assumed to be a Prolog term in canonical form, i.e., can be read by `read_canonical/1`. The corresponding value will be converted into a Prolog term and bound to `VAR`. This allows a programmer to store complex Prolog terms in a database. Variables in such a term are local only to that term.

When connecting to multiple data sources, you should use the form:

```
| ?- odbc_sql(ConnectionName,BindVals,SQLStmt,ResultRow).
```

For example, we can define a predicate, `get_test_name_price`, which given a test ID, retrieves the name and price of that test from the test table in the hospital database:

```
get_test_name_price(Id,Nam,Pri) :-
        odbc_sql([Id],'SELECT TName,Price FROM Test WHERE TId = ?', [Nam,Pri]).
```

The interface uses a cursor to retrieve this result and caches the cursor, so that if the same query is needed in the future, it does not need to be re-parsed, and re-optimized. Thus, if this predicate were to be called several times, the above form is more efficient than the following form, which must be parsed and optimized for each and every call:

```
get_test_name_price(Id,Nam,Pri) :-
      odbc_sql([],['SELECT TName,Price FROM Test WHERE TId = ''',Id,''''], [Nam,Pri]).
```

Note that to include a quote (') in an atom, it must be represented by using two quotes.

There is also a predicate:

```
| ?- odbc_sql_cnt(ConnectionName,BindVals,SQLStmt,Count).
```

This predicate is very similar to `odbc_slq/4` except that it can only be used for UPDATE, INSERT, and DELETE SQL statements. The first three arguments are just as in `odbc_slq/4`; the fourth must be a variable in which is returned the integer count of the number of rows affected by the SQL operation.

1.2.3 Cursor Management

The XSB-ODBC interface is limited to using 100 open cursors. When XSB systems use database accesses in a complicated manner, management of open cursors can be a problem due to the tuple-at-a-time access of databases from Prolog, and due to leakage of cursors through cuts and throws. Often, it is more efficient to call the database through set-at-a-time predicates such as `findall/3`, and then to backtrack through the returned information. For instance, the predicate `findall_odbc_sql/4` can be defined as:

```
findall_odbc_sql(ConnName,BindVals,SQLStmt,ResultRow):-
            findall(Res,odbc_sql(ConnName,BindVals,SQLStmt,Res),Results),
            member(ResultRow,Results).
```

As a convenience, therefore, the predicates `findall_odbc_sql/3` and `findall_odbc_sql/4` are defined in the ODBC interface.

1.2.4 Accessing Tables in Data Sources through the Relation Level

While all access to a database is possible using SQL as described above, the XSB-ODBC interface supports higher-level interaction for which the user need not know or write SQL statements; that is done as necessary by the interface. With the relation level interface, users can simply declare a predicate to access a table and the system generates the necessary underlying code, generating specialized code for each mode in which the predicate is called.

To declare a predicate to access a database table, a user must use the `odbc_import/2` interface predicate.

The syntax of `odbc_import/2` is as follows:

```
| ?- odbc_import('TableName'('FIELD1', 'FIELD2', ..., 'FIELDn'), 'PredicateName').
```

where `'TableName'` is the name of the database table to be accessed and `'PredicateName'` is the name of the XSB predicate through which access will be made. `'FIELD1'`, `'FIELD2'`, ... , `'FIELDn'` are the exact attribute names(case sensitive) as defined in the database table schema. The chosen columns define the view and the order of arguments for the database predicate `'PredicateName'`.

For example, to create a link to the `Test` table through the `'test'` predicate:

```
| ?- odbc_import('Test'('TId','TName','Length','Price'),test).

yes
```

When connecting to multiple data sources, you should use the form:

```
| ?- odbc_import(ConnectionName,
                 'TableName'('FIELD1', 'FIELD2', ..., 'FIELDn'),
                 'PredicateName').
```

1.2.5 Using the Relation Level Interface

Once the links between tables and predicates have been successfully established, information can then be extracted from these tables using the corresponding predicates. Continuing from the above example, now rows from the table `Test` can be obtained:

```
| ?- test(TId, TName, L, P).

TId = t001
TName = X-Ray
L = 5
P = 100
```

Backtracking can then be used to retrieve the next row of the table `Test`.

Records with particular field values may be selected in the same way as in Prolog; no mode specification for database predicates is required. For example:

```
| ?- test(TId, 'X-Ray', L, P).
```

will automatically generate the query:

```
SELECT rel1.TId, rel1.TName, rel1.Length, rel1.Price
FROM Test rel1
WHERE rel1.TName = ?
```

and

```
| ?- test('NULL'(_), 'X-Ray',  L, P).
```

generates: (See Section 1.2.6)

```
SELECT NULL , rel1.TName, rel1.Length, rel1.Price
FROM Test rel1
WHERE rel1.TId IS NULL AND rel1.TName = ?
```

During the execution of this query the bind variable `?` will be bound to the value `'X-Ray'`.

Of course, the same considerations about cursors noted in Section 1.2.3 apply to the relation-level interface. Accordingly, the ODBC interface also defines the predicate `odbc_import/4` which allows the user to specify that rows are to be fetched through `findall/3`. For example, the call

```
odbc_import('Test'('TId','TName','Length','Price'),test,[findall(true)]).
```

will behave as described above *but* will make all database calls through `findall/3` and return rows by backtracking through a list rather than maintaining open cursors.

Also as a courtesy to Quintus Prolog users we have provided compatibility support for some PRODBI predicates which access tables at a relational level [2].

```
| ?- odbc_attach(PredicateName, table(TableName)).
```

eg. invoke

[2]This predicate is obsolescent and `odbc_import/{2,3,4}` should be used instead.

```
| ?- odbc_attach(test2, table('Test')).
```

and then execute

```
| ?- test2(TId, TName, L, P).
```

to retrieve the rows.

1.2.6 Handling NULL values

The interface treats NULL's by introducing a single valued function `'NULL'/1` whose single value is a unique (Skolem) constant. For example a NULL value may be represented by

```
'NULL'(null123245)
```

Under this representation, two distinct NULL values will not unify. On the other hand, the search condition `IS NULL Field` can be represented in XSB as `Field = 'NULL'(_)`

Using this representation of NULL's the following protocol for queries and updates is established.

Queries

```
| ?- dept('NULL'(_),_,_).
```

Generates the query:

```
SELECT NULL , rel1.DNAME , rel1.LOC
  FROM DEPT rel1
  WHERE rel1.DEPTNO IS NULL;
```

Hence, `'NULL'(_)` can be used to retrieve rows with NULL values at any field.

`'NULL'/1` fails the predicate whenever it is used with a bound argument.

```
| ?- dept('NULL'(null2745),_,_).  → fails always.
```

Query Results

When returning NULL's as field values, the interface returns `NULL/1` function with a unique integer argument serving as a skolem constant.

Notice that the above guarantees the expected semantics for the join statements. In the following example, even if `Deptno` is NULL for some rows in `emp` or `dept` tables, the query still evaluates the join successfully.

```
| ?- emp(Ename,_,_,_,Deptno),dept(Deptno,Dname,Loc)..
```

Inserts

To insert rows with NULL values you can use `Field = 'NULL'(_)` or `Field = 'NULL'(null2346)`. For example:

```
| ?- emp_ins('NULL'(_), ...).  → inserts a NULL value for ENAME

| ?- emp_ins('NULL'('bound'), ...)  → inserts a NULL value for ENAME.
```

Deletes

To delete rows with NULL values at any particular `FIELD` use `Field = 'NULL'(_)`, `'NULL'/1` with a free argument. When `'NULL'/1` 's argument is bound it fails the delete predicate always. For example:

```
| ?- emp_del('NULL'(_), ..).  → adds ENAME IS NULL to the generated SQL
                                statement

| ?- emp_del('NULL'('bound'), ...).  → fails always
```

The reason for the above protocol is to preserve the semantics of deletes, when some free arguments of a delete predicate get bound by some preceding predicates. For example in the following clause, the semantics is preserved even if the `Deptno` field is NULL for some rows.

```
| ?- emp(_,_,_,_,Deptno), dept_del(Deptno).
```

1.2.7 The View Level Interface

The view level interface can be used to define XSB queries which include only imported database predicates (by using the relation level interface) described above and aggregate predicates (defined below). When these queries are invoked, they are translated into complex database queries, which are then executed taking advantage of the query processing ability of the DBMS.

One can use the view level interface through the predicate `odbc_query/2`:

```
| ?- odbc_query('QueryName'(ARG1, ..., ARGn), DatabaseGoal).
```

All arguments are standard XSB terms. `ARG1`, `ARG2`, ..., `ARGn` define the attributes to be retrieved from the database, while `DatabaseGoal` is an XSB goal (i.e. a possible body of a rule) that defines the selection restrictions and join conditions.

The compiler is a simple extension of [1] which generates SQL queries with bind variables and handles NULL values as described in Section 1.2.6. It allows negation, the expression of arithmetic functions, and higher-order constructs such as grouping, sorting, and aggregate functions.

Database goals are translated according to the following rules from [1]:

- Disjunctive goals translate to distinct SQL queries connected through the UNION operator.
- Goal conjunctions translate to joins.
- Negated goals translate to negated EXISTS subqueries.
- Variables with single occurrences in the body are not translated.
- Free variables translate to grouping attributes.
- Shared variables in goals translate to equi-join conditions.
- Constants translate to equality comparisons of an attribute and the constant value.
- Nulls are translated to `IS NULL` conditions.

For more examples and implementation details see [1].

In the following, we show the definition of a simple join view between the two database predicates *Room* and *Floor*.

Assuming the declarations:

```
| ?- odbc_import('Room'('RoomNo','CostPerDay','Capacity','FId'),room).

| ?- odbc_import('Floor'('FId','','FName'),floor).
```

use

```
| ?- odbc_query(query1(RoomNo,FName),
                 (room(RoomNo,_,_,FId),floor(FId,_,FName))).
yes

| ?- query1(RoomNo,FloorName).
```

Prolog/SQL compiler generates the SQL statement:

```
SELECT rel1.RoomNo , rel2.FName FROM Room rel1 , Floor rel2
WHERE rel2.FId = rel1.FId;
```

Backtracking can then be used to retrieve the next row of the view.

```
| ?- query1('101','NULL'(_)).
```

generates the SQL statement:

```
SELECT rel1.RoomNo, NULL
FROM Room rel1 , Floor rel2
WHERE rel1.RoomId = ? AND rel2.FId = rel1.FId AND rel2.FName IS NULL;
```

The view interface also supports aggregate functions such as sum, avg, count, min and max. For example

```
| ?- odbc_import('Doctor'('DId', 'FId', 'DName','PhoneNo','ChargePerMin'),doctor).

yes
| ?- odbc_query(avgchargepermin(X),
                (X is avg(ChargePerMin, A1 ^ A2 ^ A3 ^ A4 ^
                          doctor(A1,A2, A3,A4,ChargePerMin)))).

yes
```

```
| ?- avgchargepermin(X).

SELECT AVG(rel1.ChargePerMin)
FROM doctor rel1;

X = 1.64

yes
```

A more complicated example is the following:

```
| ?- odbc_query(nonsense(A,B,C,D,E),
                (doctor(A, B, C, D, E),
                 not floor('First Floor', B),
                 not (A = 'd001'),
                 E > avg(ChargePerMin, A1 ^ A2 ^ A3 ^ A4 ^
                         (doctor(A1, A2, A3, A4, ChargePerMin))))).

| ?- nonsense(A,'4',C,D,E).

SELECT rel1.DId , rel1.FId , rel1.DName , rel1.PhoneNo , rel1.ChargePerMin
FROM doctor rel1
WHERE rel1.FId = ? AND NOT EXISTS
(SELECT *
FROM Floor rel2
WHERE rel2.FName = 'First Floor' and rel2.FId = rel1.FId
) AND rel1.Did <> 'd001' AND rel1.ChargePerMin >
(SELECT AVG(rel3.ChargePerMin)
FROM Doctor rel3
);


A = d004
C = Tom Wilson
D = 516-252-100
E = 2.5
```

All database queries defined by `odbc_query/{2,3}` can be queried with any mode.

Note that at each call to a database relation or rule, the communication takes place through bind variables. The corresponding restrictive SQL query is generated, and if this is the first call with that adornment, it is cached. A second call with same adornment would

try to use the same database cursor if still available, without reparsing the respective SQL statement. Otherwise, it would find an unused cursor and retrieve the results. In this way efficient access methods for relations and database rules can be maintained throughout the session.

If connecting to multiple data sources, use the form:

```
:- odbc_query(connectionName,'QueryName'(ARG1, ..., ARGn), DatabaseGoal).
```

1.2.8 Insertions and Deletions of Rows through the Relational Level

Insertion and deletion operations can also be performed on an imported table. The two predicates to accomplish these operations are `odbc_insert/2` and `odbc_delete/2`. The syntax of `odbc_insert/2` is as follows: the first argument is the declared database predicate for insertions and the second argument is some imported data source relation. The second argument can be declared with some of its arguments bound to constants. For example after `Room` is imported through `odbc_import`:

```
|?- odbc_import('Room'('RoomNo','CostPerDay','Capacity','FId'), room).
yes
```

Now we can do

```
| ?- odbc_insert(room_ins(A1,A2,A3),(room(A1,A2,A3,'3'))).

yes
| ?- room_ins('306','NULL'(_),2).

yes
```

This will insert the row: ('306',NULL, 2,'3') into the table `Room`. Note that any call to `room_ins/7` should have all its arguments bound.

See Section 1.2.6) for information about NULL value handling.

The first argument of `odbc_delete/2` predicate is the declared delete predicate and the second argument is the imported data source relation with the condition for requested deletes, if any. The condition is limited to simple comparisons. For example assuming `Room/3` has been imported as above:

```
| ?- odbc_delete(room_del(A), (room('306',A,B,C), A > 2)).

yes
```

After this declaration you can use:

```
| ?- room_del(3).
```

to generate the SQL statement:

```
DELETE From Room rel1
WHERE rel1.RoomNo = '306' AND rel1.CostPerDay = ? AND ? > 2
;
```

Note that you have to commit your inserts or deletes to tables to make them permanent. (See section 1.2.11).

These predicates also have the form in which an additional first argument indicates a connection, for use with multiple data sources.

Also, some ODBC drivers have been found that do not accept the form of SQL generated for deletes. In these cases, you must use the lower-level interface: `odbc_sql`.

1.2.9 Access to Data Dictionaries

The following utility predicates provide users with tools to access data dictionaries [3]. A brief description of these predicates is as follows:

odbc_show_schema(accessible(Owner)) Shows the names of all accessible tables that are owned by Owner. (This list can be long!) If Owner is a variable, all tables will be shown, grouped by owner.

odbc_show_schema(user) Shows just those tables that belongs to user.

odbc_show_schema(tuples('Table')) Shows all rows of the database table named `'Table'`.

odbc_show_schema(arity('Table')) The number of fields in the table `'Table'`.

odbc_show_schema(columns('Table')) The field names of a table.

[3] Users of Quintus Prolog may note that these predicates are all PRODBI compatible.

For retrieving above information use:

- `odbc_get_schema(accessible(Owner),List)`
- `odbc_get_schema(user,List)`
- `odbc_get_schema(arity('Table'),List)`
- `odbc_get_schema(columns('Table'),List)`

The results of above are returned in List as a list.

1.2.10 Other Database Operations

odbc_create_table('TableName','FIELDs') `FIELDS` is the field specification as in SQL.

```
eg. odbc_create_table('MyTable', 'Col1 NUMBER,
                                  Col2 TEXT(50),
                                  Col3 TEXT(13)').
```

odbc_create_index('TableName','IndexName', index(_,Fields)) `Fields` is the list of columns for which an index is requested. For example:

```
odbc_create_index('Doctor', 'DocKey', index(_,'DId')).
```

odbc_delete_table('TableName') To delete a table named `'TableName'`

odbc_delete_view('ViewName') To delete a view named `'ViewName'`

odbc_delete_index('IndexName') To delete an index named `'IndexName'`

1.2.11 Transaction Management

Depending on how the transaction options are set in ODBC.INI for data sources, changes to the data source tables may not be committed (i.e., the changes become permanent) until the user explicitly issues a commit statement. Some ODBC drivers support autocommit, which, if on, means that every update operation is immediately committed upon execution. If autocommit is off, then an explicit commit (or rollback) must be done by the program to ensure the updates become permanent (or are ignored.).

The predicate `odbc_transaction/1` supports these operations.

odbc_transaction(autocommit(on)) Turns on autocommit, so that all update operations will be immediately committed on completion.

odbc_transaction(autocommit(off)) Turns off autocommit, so that all update operations will not be committed until explicitly done so by the program (using one of the following operations.)

odbc_transaction(commit) Commits all transactions up to this point. (Only has an effect if autocommit is off).

odbc_transaction(rollback) Rolls back all update operations done since the last commit point. (Only has an effect if autocommit is off).

1.2.12 Interface Flags

Users are given the option to monitor control aspects of the ODBC interface by setting ODBC flags via the predicates`set_odbc_flag/2` and `odbc_flag/2`.

The first aspect that can be controlled is whether to display SQL statements for SQL queries. This is done by the `show_query` flag. For example:

```
| ?- odbc_flag(show_query,Val).

     Val = on
```

Indicates that SQL statements will now be displayed for all SQL queries, and is the default value for the ODBC interface. To turn it off execute the command `set_odbc_flag(show_query,on)`.

The second aspect that can be controlled is the action taken upon ODBC errors. Three possible actions may be useful in different contexts and with different drivers. First, the error may be ignored, so that a database call succeeds; second the error cause the predicate to fail, and third the error may cause an exception to be thrown to be handled by a catcher (or the default system error handler, see Volume 1).

`| ?- odbc_flag(fail_on_error, ignore)` Ignores all ODBC errors, apart from writing a warning. In this case, it's the users' users' responsibility to check each of their actions and do error handling.

`| ?- odbc_flag(fail_on_error, fail)` Interface fails whenever error occurs.

`| ?- odbc_flag(fail_on_error, throw)` Throws an error-term of the form `error(odbc_error,Message` in which `Message` is a textual description of the ODBC error, and `Backtrace` is a list of the continuations of the call. These continuations may be printed out by the error handler.

The default value of fail_on_error is on.

1.2.13 Datalog

Users can write recursive Datalog queries with exactly the same semantics as in XSB using imported database predicates or database rules. For example assuming `odbc_parent/2` is an imported database predicate, the following recursive query computes its transitive closure.

```
:- table(ancestor/2).
ancestor(X,Y) :- odbc_parent(X,Y).
ancestor(X,Z) :- ancestor(X,Y), odbc_parent(Y,Z).
```

This works with drivers that support multiple open cursors to the same connection at the same time. (Sadly, some don't.) In the case of drivers that don't support multiple open cursors, one can often replace each odbc_import-ed predicate call

```
...,predForTable(A,B,C),...
```

by

```
...,findall([A,B,C],predForTable(A,B,C),PredList),
    member([A,B,C],PredList)...
```

and get the desired effect.

1.3 Error messages

ERR - DB: Connection failed For some reason the attempt to connect to data source failed.

- Diagnosis: Try to see if the data source has been registered with Microsoft ODBC Administrator, the username and password are correct and MAXCURSORNUM is not set to a very large number.

ERR - DB: Parse error The SQL statement generated by the Interface or the first argument to `odbc_sql/1` or `odbc_sql_select/2` can not be parsed by the data source driver.

- Diagnosis: Check the SQL statement. If our interface generated the erroneous statement please contact us at `xsb-contact@cs.sunysb.edu`.

ERR - DB: No more cursors left Interface run out of non-active cursors either because of a leak or no more free cursors left.

- Diagnosis: System fails always with this error. odbc_transaction(rollback) or odbc_transaction(commit) should resolve this by freeing all cursors.

ERR - DB: FETCH failed Normally this error should not occur if the interface running properly.

- Diagnosis: Please contact us at xsb-contact@cs.sunysb.edu

1.4 Notes on specific ODBC drivers

MyODBC The ODBC driver for MySQL is called MyODBC, and it presents some particularities that should be noted.

First, MySQL, as of version 3.23.55, does not support strings of length greater than 255 characters. XSB's ODBC interface has been updated to allow the use of the BLOB datatype to encode larger strings.

More importantly, MyODBC implements SQLDescribeCol such that, by default, it returns actual lengths of columns in the result table, instead of the formal lengths in the tables. For example, suppose you have, in table A, a field f declared as "VARCHAR (200)". Now, you create a query of the form "SELECT f FROM A WHERE ..." If, in the result set, the largest size of f is 52, that's the length that SQLDescribeCol will return. This breaks XSB's caching of query-related data-structures. In order to prevent this behavior, you should configure your DSN setup so that you pass "Option=1" to MyODBC.

Chapter 2

The New XSB-Database Interface

By Saikat Mukherjee, Michael Kifer and Hui Wan

2.1 Introduction

The XSB-DB interface is a package that allows XSB users to access databases through various drivers. Using this interface, information in different DBMSs can be accessed by SQL queries. The interface defines Prolog predicates which makes it easy to connect to databases, query them, and disconnect from the databases. Central to the concept of a connection to a database is the notion of a *connection handle*. A connection handle describes a particular connection to a database. Similar to a connection handle is the notion of a query handle which describes a particular query statement. As a consequence of the handles, it is possible to open multiple database connections (to the same or different databases) and keep alive multiple queries (again from the same or different connections). The interface also supports dynamic loading of drivers. As a result, it is possible to query databases using different drivers concurrently [1].

Currently, this package provides drivers for ODBC, a native MySQL driver, and a driver for the embedded MySQL server.

2.2 Configuring the Interface

Generally, each driver has to be configured separately, but if the database packages such as ODBC, MySql, etc., are installed in standard places then the XSB configuration mechanism will do the job automatically.

[1] In Version 4.0, this package has not been ported to the multi-threaded engine.

Under Windows, first make sure that XSB is configured and built correctly for Windows, and that it runs. As part of that building process, the command

```
makexsb_wind
```

must have been executed in the directory `XSB\build`. It will normally configure the ODBC driver without problems. For the MySQL driver one has to edit the file

```
packages\dbdrivers\mysql\cc\NMakefile.mak
```

to indicate where MySQL is installed. To build the embedded MySQL driver under Windows, the file

```
packages\dbdrivers\mysqlenbedded\cc\NMakefile.mak
```

might need to be edited. Then you should either rebuild XSB using the `makexsb_wind` command or by running

```
nmake /f NMakefile.mak
```

in the appropriate directories (`dbdrivers\mysql\cc` or `dbdrivers\mysqlenbedded\cc`). Note that you need a C++ compiler and `nmake` installed on your system for this to work.[2]

Under Unix, the `configure` script will build the drivers automatically if the `-with-dbdrivers` option is specified. If, however, ODBC and MySQL are not installed in their standard places, you will have to provide the following parameters to the configure script:

- `-with-odbc-libdir=LibDIR` – `LibDIR` is the directory where the library libodbc.so lives on your system.
- `-with-odbc-incdir=IncludeDIR` – `IncludeDIR` is the directory where the ODBC header files, such as `sql.h` live.
- `-with-mysql-libdir=MySQLlibdir` – `MySQLlibdir` is the directory where MySQL's shared libraries live on your system.
- `-with-mysql-incdir=MySQLincludeDir` – `MySQLincludeDir` is the directory where MySQL's header files live.

If you are also using the embedded MySQL server and want to take advantage of the corresponding XSB driver, you need to provide the following directories to tell XSB where the copy of MySQL that supports the embedded server is installed. This has to be done *only* if that copy is not in a standard place, like `/usr/lib/mysql`.

[2] `http://www.microsoft.com/express/vc/`
`http://download.microsoft.com/download/vc15/Patch/1.52/W95/EN-US/Nmake15.exe`

- `-with-mysqlembedded-libdir=MySQLlibdir` – `MySQLlibdir` is the directory where MySQL's shared libraries live on your system. This copy of MySQL must be configured with support for the embedded server.
- `-with-mysqlembedded-incdir=MySQLincludeDir` – `MySQLincludeDir` is the directory where MySQL's header files live.

Under Cygwin, the ODBC libraries come with the distribution; they are located in the directory `/cygdrive/c/cygwin/lib/w32api/` and are called `odbc32.a` and `odbccp32.a`. (Check if your installation is complete and has these libraries!) Otherwise, the configuration of the interface under Cygwin is same as in unix (you do not need to provide any ODBC-specific parameters to the configure script under Cygwin).

If at the time of configuring XSB some database packages (*e.g.*, MySQL) are not installed on your system, you can install them later and configure the XSB interface to them then. For instance, to configure the ODBC interface separately, you can type

```
cd packages/dbdrivers/odbc
configure
```

Again, if ODBC is installed in a non-standard location, you might need to supply the options `-with-odbc-libdir` and `-with-odbc-incdir` to the configure script. Under Cygwin ODBC is always installed in a standard place, and `configure` needs no additional parameters.

Under Windows, separate configuration of the XSB-DB interfaces is also possible, but you need Visual Studio installed. For instance, to configure the MySQL interface, type

```
cd packages\dbdrivers\mysql\cc
nmake /f NMakefile.mak
```

As before, you might need to edit the NMakefile.mak script to tell the compiler where the required MySQL's libraries are. You also need the file `packages\dbdrivers\mysql\mysql_init.P` with the following content:

```
:- export mysql_info/2.
mysql_info(support, 'yes').
mysql_info(libdir,  '').
mysql_info(ccflags, '').
mysql_info(ldflags, '').
```

Similarly, to configure the ODBC interface, do

```
cd packages\dbdrivers\odbc\cc
nmake /f NMakefile.mak
```

You will also need to create the file `packages\dbdrivers\odbc\odbc_init.P` with the following contents:

```
:- export odbc_info/2.
odbc_info(support, 'yes').
odbc_info(libdir,  '').
odbc_info(ccflags, '').
odbc_info(ldflags, '').
```

2.3 Using the Interface

We use the `student` database as our example to illustrate the usage of the XSB-DB interface in this manual. The schema of the student database contains three columns viz. the student name, the student id, and the name of the advisor of the student.

The XSB-DB package has to be first loaded before using any of the predicates. This is done by the call:

```
| ?- [dbdrivers].
```

Next, the driver to be used for connecting to the database has to be loaded. Currently, the interface has support for a native MySQL driver (using the MySQL C API), and an ODBC driver. For example, to load the ODBC driver call:

```
| ?- load_driver(odbc).
```

Similarly, to load the mysql driver call:

```
| ?- load_driver(mysql).
```

or

```
| ?- load_driver(mysqlembedded).
```

2.3.1 Connecting to and Disconnecting from Databases

There are two predicates for connecting to databases, `db_connect/5` and `db_connect/6`. The `db_connect/5` predicate is for ODBC connections, while `db_connect/6` is for other (non-ODBC) database drivers.

```
| ?- db_connect(+Handle, +Driver, +DSN, +User, +Password).
| ?- db_connect(+Handle, +Driver, +Server, +Database, +User, +Password).
```

The `db_connect/5` predicate assumes that an entry for a data source name (DSN) exists in the `odbc.ini` file. The `Handle` is the connection handle name used for the connection. The `Driver` is the driver being used for the connection. The `User` and `Password` are the user name and password being used for the connection. The user is responsible for giving the name to the handle. To connect to the data source mydb using the user name xsb and password xsb with the odbc driver, the call is as follows:

```
| ?- db_connect(ha, odbc, mydb, xsb, xsb).
```

where `ha` is the user-chosen handle name (a Prolog atom) for the connection.

The `db_connect/6` predicate is used for drivers other than ODBC. The arguments `Handle`, `Driver`, `User`, and `Password` are the same as for `db_connect/5`. The `Server` and `Database` arguments specify the server and database to connect to. For example, for a connection to a database called `test` located on the server `wolfe` with the user name `xsb`, the password `foo`, and using the `mysql` driver, the call is:

```
| ?- db_connect(ha, mysql, wolfe, test, xsb, foo).
```

where `ha` is the handle name the user chose for the connection.

If the connection is successfully made, the predicate invocation will succeed. This step is necessary before anything can be done with the data sources since it gives XSB the opportunity to initialize system resources for the session.

To close a database connection use:

```
| ?- db_disconnect(Handle).
```

where handle is the connection handle name. For example, to close the connection to above mysql database call:

```
| ?- db_disconnect(ha).
```

and XSB will give all the resources it allocated for this session back to the system.

2.3.2 Querying Databases

The interface supports two types of querying. In direct querying, the query statement is not prepared while in prepared querying the query statement is prepared before being executed. The results from both types of querying are retrieved tuple at a time. Direct querying is done by the predicate:

```
| ?- db_query(ConnectionHandle, QueryHandle, SQLQueryList, ReturnList).
```

ConnectionHandle is the name of the handle used for the database connection. QueryHandle is the name of the *query handle* for this particular query. For prepared queries, the query handle is used both in order to execute the query and to close it and free up space. For direct querying, the query handle is used only for closing query statements (see below). The `SQLQueryList` is a list of terms which is used to build the SQL query. The terms in this list can have variables, which can be instantiated by the preceding queries. The query list is scanned for terms, which are encoded into Prolog atoms and the result is then concatenated; it must form a valid SQL query. (The treatment of terms is further discussed below.) `ReturnList` is a list of variables each of which correspond to a return value in the query. It is upto the user to specify the correct number of return variables corresponding to the query. Also, as in the case of a connection handle, the user is responsible for giving the name to the query handle. For example, a query on the student database to select all the students for a given advisor is accomplished by the call:

```
| ?- X = adv,
     db_query(ha,qa,['select T.name from student T where T.advisor=',X],[P]),
     fail.
```

where `ha` and `qa` are respectively the connection handle and query handle name the user chose.

Observe that the query list is composed of the SQL string and a ground value for the advisor. The return list is made of one variable corresponding to the student name. The failure drive loop retrieves all the tuples.

Preparing a query is done by calling the following predicate:

```
| ?- db_prepare(ConnectionHandle, QueryHandle, SQLQueryList).
```

As before, ConnectionHandle and QueryHandle specify the handles for the connection and the query. The SQLQueryList is a list of terms which build up the query string. The placeholder '?' is used for values which have to be bound during the execution of the statement. For example, to prepare a query for selecting the advisor name for a student name using our student database:

```
| ?- db_prepare(ha,qa,['select T.advisor from student T where T.name = ?']).
```

A prepared statement is executed using the predicate:

```
| ?- db_prepare_execute(QueryHandle, BindList, ReturnList).
```

The BindList contains the ground values corresponding to the '?' in the prepared statement. The ReturnList is a list of variables for each argument in a tuple of the result set. For instance,

```
| ?- db_prepare_execute(qa,['Bob'],[?Advisor]).
```

For direct querying, the query handle is closed automatically when all the tuples in the result set have been retrieved. In order to explicitly close a query handle, and free all the resources associated with the handle, a call is made to the predicate:

```
| ?- db_statement_close(QueryHandle).
```

where QueryHandle is the query handle for the statement to be closed.

Storing and retrieving terms and NULL values. The interface is also able to transparently handle Prolog terms. Users can both save and retrieve terms in string fields of the tables by passing the term as a separate element in the query list and making sure that it is enclosed in quotes in the concatenated result. For instance,

```
?- db_query(handle,qh,['insert into mytbl values(11,22,',p(a),')'],[]).
```

The above statement inserts `p(a)` *as a term* into the third column of the table `mytbl`. Under the hood, it is inserted as a special string, but when retrieved, this term is decoded back into a Prolog term. For this to work, the third column of `mytbl` *must* be declared as a character string (e.g., `CHAR(50)`). Important to note is that `p(a)` has to appear as a list element above and not be quoted so that Prolog will recognize it as a term.

The `NULL` value is represented using the special 0-ary term `'NULL'(_)` *when retrieved*. When you need to *store* a null value, you can use either the above special term or just place `NULL` in the appropriate place in the SQL INSERT statement. For instance,

```
?- db_query(handle,qh1,['insert into mytbl values(11,22,NULL)'],[]).
?- db_query(handle,qh2,['insert into mytbl values(111,222,','NULL'(),')'],[]).
```

However, when retrieved from a database, a `NULL` is always represented by the term `'NULL'(_)` (and not by the atom `'NULL'`).

2.4 Error Handling

Each predicate in the XSB-DB interface throws an exception with the functor

```
dbdrivers_error(Number, Message)
```

where Number is a string with the error number and Message is a string with a slightly detailed error message. It is upto the user to catch this exception and proceed with error handling. This is done by the throw-catch error handling mechanism in XSB. For example, in order to catch the error which will be thrown when the user attempts to close a database connection for a handle (`ha`) which does not exist:

```
| ?- catch(db_disconnect(ha),
     dbdrivers_error(Number, Message), handler(Number, Message)).
```

It is the user's responsibility to define the handler predicate which can be as simple as printing out the error number and message or may involve more complicated processing.

A list of error numbers and messages that are thrown by the XSB-DB interface is given below:

- **XSB_DBI_001: Driver already registered**
  This error is thrown when the user tries to load a driver, using the `load_driver` predicate, which has already been loaded previously.
- **XSB_DBI_002: Driver does not exist**
  This error is thrown when the user tries to connect to a database, using `db_connect`, with a driver which has not been loaded.
- **XSB_DBI_003: Function does not exist in this driver**
  This error is thrown when the user tries to use a function support for which does not exist in the corresponding driver. For example, this error is generated if the user tries to use `db_prepare` for a connection established with the mysql driver.
- **XSB_DBI_004: No such connection handle**
  This error is thrown when the user tries to use a connection handle which has not been created.
- **XSB_DBI_005: No such query handle**
  This error is thrown when the user tries to use a query handle which has not been created.
- **XSB_DBI_006: Connection handle already exists**
  This error is thrown when the user tries to create a connection handle in `db_connect` using a name which already exists as a connection handle.

- **XSB_DBI_007: Query handle already exists**
  This error is thrown when the user tries to create a query handle, in `db_query` or `db_prepare`, using a name which already exists as a query handle for a different query.

- **XSB_DBI_008: Not all parameters supplied**
  This error is thrown when the user tries to execute a prepared statement, using `db_prepare_execute`, without supplying values for all the parameters in the statement.

- **XSB_DBI_009: Unbound variable in parameter list**
  This error is thrown when the user tries to execute a prepared statement, using `db_prepare_execute`, without binding all the parameters of the statement.

- **XSB_DBI_010: Same query handle used for different queries**
  This error is thrown when the user issues a prepare statement (`db_prepare`) using a query handle that has been in use by another prepared statement and which has not been closed. Query handles must be closed before reuse.

- **XSB_DBI_011: Number of requested columns exceeds the number of columns in the query**
  This error is thrown when the user `db_query` specifies more items to be returned in the last argument than the number of items in the `SELECT` statement in the corresponding query.

- **XSB_DBI_012: Number of requested columns is less than the number of columns in the query**
  This error is thrown when the user `db_query` specifies fewer items to be returned in the last argument than the number of items in the `SELECT` statement in the corresponding query.

- **XSB_DBI_013: Invalid return list in query**
  Something else is wrong with the return list of the query.

- **XSB_DBI_014: Too many open connections**
  There is a limit (200) on the number of open connections.

- **XSB_DBI_015: Too many registered drivers**
  There is a limit (100) on the number of database drivers that can be registered at the same time.

- **XSB_DBI_016: Too many active queries**
  There is a limit (2000) on the number of queries that can remain open at any given time.

2.5 Notes on specific drivers

Note: in most distributions of Linux, with all of these drivers you need to install both the runtime version of the corresponding packages as well as the development version. For instance, for the `unixodbc` driver, these packages will typically have the names `unixodbc` and `unixodbc-dev`. For the MySQL driver, the packages would typically be named `libmysqlclient` and `libmysqlclient-dev`. For the embedded MySQL driver, the relevant package would be `libmysqld-pic` and `libmysqld-dev`.

ODBC Driver

The ODBC driver has been tested in Linux using the `unixodbc` driver manager. It currently supports the following functionality: (a) connecting to a database using a DSN, (b) direct querying of the database, (c) using prepared statements to query the database, (d) closing a statement handle, and (d) disconnecting from the database. The ODBC driver has also been tested under Windows and Cygwin.

MySQL Driver

The MySQL driver provides access to the native MySQL C API. Currently, it has support for the following functionality: (a) connecting to a database using `db_connect`, (b) direct querying of the database, (c) using prepared statements to query the database, (d) closing a statement handle, and (e) disconnecting from the database.

The MySQL driver has been tested under Linux and Windows.

Driver for the Embedded MySQL Server

This driver provides access to the Embedded MySQL Server Library `libmysqld`. Currently, it has support for the following functionality: (a) connecting to a database `db_connect`, (b) direct querying of the database, (c) using prepared statements to query the database, (d) closing a statement handle, and (e) disconnecting from the database.

The MySQL driver for Embedded MySQL Server has been tested under Linux.

In order to use this driver, you will need:

- MySQL with Embedded Server installed on your machine. If your don't have a precompiled binary distribution of MySQL, which was configured with libmysqld support (the embedded server library), you will need to build MySQL from sources and configure it with the `-with-embedded-server` option.

- append to `/etc/my.cnf` (or `/etc/mysql/my.cnf` – whichever is used on your machine) or `~/.my.cnf`:

```
[mysqlembedded_driver_SERVER]
language = /usr/share/mysql/english
datadir = .....
```

  You will probably need to replace `/usr/share/mysql/english` with a directory appropriate for your MySQL installation.

  You might also need to set the `datadir` option to specify the directory where the databases managed by the embedded server are to be kept. This has to be done if there is a possibility of running the embedded MySQL server alongside the regular MySQL server. In that case, the `datadir` directory of the embedded server must be different from the `datadir` directory of the regular server (which is likely to be specified using the `datadir` option in `/etc/my.cnf` or `/etc/mysql/my.cnf`. This is because specifying the same directory might lead to a corruption of your databases. See `http://dev.mysql.com/doc/refman/5.1/en/multiple-servers.html` for further details on running multiple servers.

Please note that loading the embedded MySQL driver increases the memory footprint of XSB. This additional memory is released automatically when XSB exits. If you need to release the memory before exiting XSB, you can call `driverMySQLEmbedded_lib_end` after disconnecting from MySQL. Note that once `driverMySQLEmbedded_lib_end` is called, no further connections to MySQL are allowed from the currently running session of XSB (or else XSB will exit abnormally).

Chapter 3

Libraries from Other Prologs

XSB is distributed with some libraries that have been provided from other Prologs.

3.1 AVL Trees

By Mats Carlsson

AVL trees (i.e., triees subject to the Adelson-Velskii-Landis balance criterion) provide a mechanism to maintain key value pairs so that loop up, insertion, and deletion all have complexity $\mathcal{O}(\log n)$. This library contains predicates to transform a sorted list to an AVL tree and back, along with predicates to manipulate the AVL trees [1]

`list_to_assoc(+List, ?Assoc)` `module: list_to_assoc/2`
: is true when `List` is a proper list of Key-Val pairs (in any order) and `Assoc` is an association tree specifying the same finite function from Keys to Values.

`assoc_to_list(+Assoc, ?List)` `module: assoc_to_list/2`
: assoc assumes that `Assoc` is a proper AVL tree, and is true when `List` is a list of Key-Value pairs in ascending order with no duplicate keys specifying the same finite function as `Assoc`. Use this to convert an `Assoc` to a list.

`assoc_vals_to_list(+Assoc, ?List)` `module: assoc_vals_to_list/2`
: assoc assumes that `Assoc` is a proper AVL tree, and is true when `List` is a list of Values in ascending order of Key with no duplicate keys specifying the same finite function as `Assoc`. Use this to extract the list of Values from `Assoc`.

`is_assoc(+Assoc)` `module: is_assoc/1`
: assoc is true when `Assoc` is a (proper) AVL tree. It checks both that the keys are in ascending order and that `Assoc` is properly balanced.

[1] This library contains functionality not documented here: see the code file for further documentation.

`gen_assoc(?Key, +Assoc, ?Value)` `module: gen_assoc/3`
assoc assumes that `Assoc` is a proper AVL tree, and is true when Key is associated with Value in `Assoc`. Can be used to enumerate all Values by ascending Keys.

`get_assoc(+Key, +OldAssoc,?OldValue,?NewAssoc,?NewValue)` `module: get_assoc/5`
assoc is true when `OldAssoc` and `NewAssoc` are AVL trees of the same shape having the same elements except that the value for `Key` in `OldAssoc` is `OldValue` and the value for `Key` in `NewAssoc` is `NewValue`.

`put_assoc(+Key,+OldAssoc,+Val,-NewAssoc)` `module: put_assoc/4`
assoc is true when `OldAssoc` and `NewAssoc` define the same finite function except that `NewAssoc` associates `Val` with `Key`. `OldAssoc` need not have associated any value at all with `Key`.

`del_assoc(+Key,+OldAssoc,?Val,-NewAssoc)` `module: del_assoc/4`
assoc is true when `OldAssoc` and `NewAssoc` define the same finite function except that `OldAssoc` associates `Key` with `Val` and `NewAssoc` doesn't associate `Key` with any value.

3.2 Unweighted Graphs: `ugraphs.P`

By Mats Carlsson

XSB also includes a library for unweighted graphs. This library allows for the representation and manipulation of directed and non-directed unlabelled graphs, including predicates to find the transitive closure of a graph, maximal paths, minimal paths, and other features. This library represents graphs as an ordered set of their edges and does not use tabling. As a result, it may be slower for large graphs than similar predicates based on a datalog representatoin of edges.

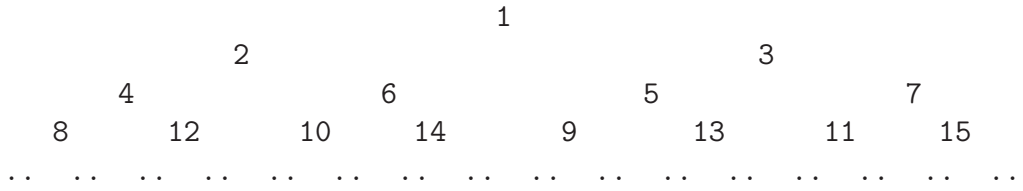
3.3 Heaps: `heaps.P`

By Richard O'Keefe

(Summary from code documentation). A heap is a labelled binary tree where the key of each node is less than or equal to the keys of its sons. The point of a heap is that we can keep on adding new elements to the heap and we can keep on taking out the minimum element. If there are N elements total, the total time is $\mathcal{O}(Nlg(N))$. If you know all the elements in advance, you are better off doing a merge-sort, but this file is for when you want to do say a best-first search, and have no idea when you start how many elements there will be, let alone what they are.

A heap is represented as a triple `t(N, Free, Tree)` where `N` is the number of elements in the tree, `Free` is a list of integers which specifies unused positions in the tree, and `Tree` is

a tree made of t terms for empty subtrees and `t(Key,Datum,Lson,Rson)` terms for the rest The nodes of the tree are notionally numbered like this:



The idea is that if the maximum number of elements that have been in the heap so far is M, and the tree currently has K elements, the tree is some subtreee of the tree of this form having exactly M elements, and the Free list is a list of $K - M$ integers saying which of the positions in the M-element tree are currently unoccupied. This free list is needed to ensure that the cost of passing N elements through the heap is $\mathcal{O}(Nlg(M))$ instead of $\mathcal{O}(NlgN)$. For M say 100 and N say 10^4 this means a factor of two.

Chapter 4

Introduction to XSB Packages

An XSB package is a piece of software that extends XSB functionality but is not critical to programming in XSB. Around a dozen packages are distributed with XSB, ranging from simple meta-interpreters to complex software systems. Some packages provide interfaces from XSB to other software systems, such as Perl, SModels or Web interfaces (as in the `libwww` package). Others, such as the CHR and Flora packages, extend XSB to different programming paradigms.

Each package is distributed in the `$XSB_DIR/packages` subdirectory, and has two parts: an initialization file, and a subdirectory in which package source code files and executables are kept. For example, the `xsbdoc` package has files `xsbdoc.P`, `xsbdoc.xwam`, and a subdirectory, `xsbdoc`. If a user doesn't want to retain `xsbdoc` (or any other package) he or she may simply remove the initialization files and the associated subdirectory without affecting the core parts of the XSB system.

Chapter 5

Wildcard Matching

By Michael Kifer

XSB has an efficient interface to POSIX wildcard matching functions. To take advantage of this feature, you must build XSB using a C compiler that supports POSIX 2.0 (for wildcard matching). This includes GCC and probably most other compilers. This also works under Windows, provided you install CygWin and use GCC to compile [1].

The `wildmatch` package provides the following functionality:

1. Telling whether a wildcard, like the ones used in Unix shells, match against a given string. Wildcards supported are of the kind available in tcsh or bash. Alternating characters (*e.g.*, "`[abc]`" or "`[^abc]`") are supported.
2. Finding the list of all file names in a given directory that match a given wildcard. This facility generalizes `directory/2` (in module `directory`), and it is much more efficient.
3. String conversion to lower and upper case.

To use this package, you need to type:

```
| ?- [wildmatch].
```

If you are planning to use it in an XSB program, you need this directive:

```
:- import glob_directory/4, wildmatch/3, convert_string/3 from wildmatch.
```

The calling sequence for `glob_directory/4` is:

[1] This package has not yet been ported to the multi-threaded engine.

```
glob_directory(+Wildcard, +Directory, ?MarkDirs, -FileList)
```

The parameter `Wildcard` can be either a Prolog atom or a Prolog string. `Directory` is also an atom or a string; it specifies the directory to be globbed. `MarkDirs` indicates whether directory names should be decorated with a trailing slash: if `MarkDirs` is bound, then directories will be so decorated. If MarkDirs is an unbound variable, then trailing slashes will not be added.

`FileList` gets the list of files in `Directory` that match `Wildcard`. If `Directory` is bound to an atom, then `FileList` gets bound to a list of atoms; if `Directory` is a Prolog string, then `FileList` will be bound to a list of strings as well.

This predicate succeeds is at least one match is found. If no matches are found or if `Directory` does not exist or cannot be read, then the predicate fails.

The calling sequence for `wildmatch/3` is as follows:

```
wildmatch(+Wildcard, +String, ?IgnoreCase)
```

`Wildcard` is the same as before. `String` represents the string to be matched against `Wildcard`. Like `Wildcard`, `String` can be an atom or a string. `IgnoreCase` indicates whether case of letters should be ignored during matching. Namely, if this argument is bound to a non-variable, then the case of letters is ignored. Otherwise, if `IgnoreCase` is a variable, then the case of letters is preserved.

This predicate succeeds when `Wildcard` matches `String` and fails otherwise.

The calling sequence for `convert_string/3` is as follows:

```
convert_string(+InputString, +OutputString, +ConversionFlag)
```

The input string must be an atom or a character list. The output string must be unbound. Its type will be "atom" if so was the input and it will be a character list if so was the input string. The conversion flag must be the atom `tolower` or `toupper`.

This predicate always succeeds, unless there was an error, such as wrong type argument passed as a parameter.

Chapter 6

pcre: Pattern Matching and Substitution Using PCRE

By Mandar Pathak

6.1 Introduction

This package employs the PCRE library to enable XSB perform pattern matching and string substitution based on Perl regular expressions.

6.2 Pattern matching

The `pcre` package provides two ways of doing pattern matching: first-match mode and bulk-match mode. The syntax of the `pcre:match/4` predicate is:

```
?- pcre:match(+Pattern, +Subject, -MatchList, +Mode).
```

To find only the first match, the `Mode` parameter must be set to the atom `one`. To find all matches, the `Mode` parameter is set to the atom `bulk`. The result of the matching is returned as a list of terms of the form

`match`(*Match*,*Prematch*,*Postmatch*,[*Subpattern1, Subpattern2,...*])

The `Pattern` and the `Subject` arguments of `pcre:match` must be XSB atoms. If there is a match in the subject, then the result is returned as a list of the `match(...)`-elements shown above. *Match* refers to the substring that matched the entire pattern. *Prematch* contains

part of the subject-string that precedes the matched substring. *Postmatch* contains part of the subject following the matched substring. The list of subpatterns (the 4-th argument of the `match` data structure) corresponds to the substrings that matched the parenthesized expressions in the given pattern. For example:

```
?- pcre:match('(\d{5}-\d{4})\ [A-Z]{2}',
'Hello12345-6789 NYwalk', X, one).
X = [match(12345-6789 NY,Hello,walk,[12345-6789])]
```

In this example, the mode argument is `one` so only one match is returned, the match found for the substring '12345-6789 NY'. The prematch is 'Hello' and the postmatch is 'walk'. The substring *'12345-6789'* matched the parenthesized expression ($\backslash d\{5\}-\backslash d\{4\}$) and hence it is returned as part of the subpatterns list.

Consider another example, one where all matches are returned:

```
?- pcre:match('[a-z]+@[a-z]+\.(com|net|edu)',
              'a@b.com@c.net@d.edu', X, bulk).
X = [match(a@b.com,,@c.net@d.edu,[com]),
     match(com@c.net,a@b.,@d.edu,[net]),
     match(om@c.net,a@b.c,@d.edu,[net]),
     match(m@c.net,a@b.co,@d.edu,[net]),
     match(net@d.edu,a@b.com@c.,,[edu]),
     match(et@d.edu,a@b.com@c.n,,[edu]),
     match(t@d.edu,a@b.com@c.ne,,[edu])]
```

This example uses the bulk match mode of the `pcre_match/4` predicate to find all possible matches that resemble a very basic email address. In case there is no prematch or postmatch to a matched substring, an empty string is returned.

In general, there can be any number of parenthesized sub-patterns in a given pattern and the subpattern match-list in the 4-th argument of the `match` data structure can have 0, 1, 2, or more elements.

6.3 String Substitution

The `pcre` package also provides a way to perform string substitution via the `pcre:substitute/4` predicate. It has the following syntax:

```
?- pcre:substitute(+Pattern, +Subject, +Substitution, -Result).
```

Pattern is the regular expression against which *Subject* is matched. Each match found is then replaced by the *Substitution*, and the result is returned in the variable *Result*. Here, *Pattern*, *Subject* and *Substitution* have to be XSB atoms whereas *Result* must be an unbound variable. The following example illustrates the use of this predicate:

```
?- pcre:substitute(is,'This is a Mississippi issue', was, X).
X = Thwas was a Mwasswassippi wassue
```

Note that the predicate `pcre:substitute/4` always works in the bulk mode. If one needs to substitute only *one* occurrence of a pattern, this is easy to do using the `pcre:match/4` predicate. For instance, if one wants to replace the third occurrence of "`is`" in the above string, we could issue the query

```
?- pcre:match(is,'This is a Mississippi issue',X,bulk).
```

take the third element in the returned list, i.e.,

```
match(is,'This is a M','sissippi issue',[])
```

and then concatenate the *Prematch* in the above `match(...)` (i.e., `'This is a M'`) with the substitute string (i.e., `'was'`) and the *Postmatch* (i.e., `'sissippi issue'`).

Additional examples of the use of the `pcre` package can be found in the XSB distribution, in the file `$XSBDIR/examples/pcretest.P`.

6.4 Installation and configuration

XSB's `pcre` package requires that the `PCRE` library is installed. For Windows, the `PCRE` library files are included with the XSB installation. For Linux and Mac, the PCRE and the PCRE-development packages must be installed using the distribution's package manager. The names of these packages might differ from one Linux distribution to the next. For instance, in Ubuntu, these libraries might be called *libpcre3* and `libpcre3-dev`. In contrast, Fedora uses the names `pcre` and `pcre-devel`. On the Mac, these packages live in the Homebrew add-on, which must be installed separately.

6.4.1 Configuring for Linux, Mac, and other Unices

In the unlikely case that your Linux distribution does not include PCRE as a package they must be downloaded and built manually. Please visit

http://www.pcre.org/

to download the latest distribution and follow the instructions given with the package.

To configure `pcre` on Linux, Mac, or on some other Unix variant, switch to the `XSB/build` directory and type:

```
cd ../packages/pcre
./configure
./makexsb
```

6.4.2 Configuring for Windows

If your installation of XSB is not configured with PCRE, you will need Microsoft `nmake` installed. Change to the top XSB directory and type:

```
cd packages\pcre\cc
nmake /f NMakefile.mak      <-- if you have the 32 bit version of XSB
nmake /f NMakefile64.mak    <-- if you have the 64 bit version of XSB
```

This builds the DLL required by XSB's `pcre` package on Windows. To make sure that the build went ahead smoothly, open the directory

```
{XSB_DIR}\config\x86-pc-windows\bin   <-- if using the 32 bit XSB
{XSB_DIR}\config\x64-pc-windows\bin   <-- if using the 64 bit XSB
```

and verify that the file `pcre4pl.dll` exists there.

Once the package has been configured, it must be loaded before it can be used:

```
?- [pcre].
```

Chapter 7

POSIX Regular Expression and Wildcard Matching

By Michael Kifer

XSB has an efficient interface to POSIX pattern regular expression and wildcard matching functions. To take advantage of these features, you must build XSB using a C compiler that supports POSIX 1.0 (for regular expression matching) and the forthcoming POSIX 2.0 (for wildcard matching). The recent versions of GCC and SunPro compiler will do, as probably will many other compilers. This also works under Windows, provided you install CygWin and use GCC to compile [1].

7.1 `regmatch`: Regular Expression Matching and Substitution

The following discussion assumes that you are familiar with the syntax of regular expressions and have a reasonably good idea about their capabilities. One easily accessible description of POSIX regular expressions is found in the on-line Emacs manual.

The regular expression matching functionality is provided by the package called `Regmatch`. To use it interactively, type:

```
:- [regmatch].
```

If you are planning to use pattern matching from within an XSB program, then you need to include the following directive:

[1] This package has not yet been ported to the multi-threaded engine.

```
:- import re_match/5, re_bulkmatch/5,
         re_substitute/4, re_substring/4
   from regmatch.
```

Matching. The predicates `re_match/5` and `re_bulkmatch/5` perform regular expression matching. The predicate `re_substitute/4` replaces substrings in a list with strings from another list and returns the resulting new string.

The `re_match/5` predicate has the following calling sequence:

```
re_match(+Regexp, +InputStr, +Offset, ?IgnoreCase, -MatchList)
```

`Regexp` is a regular expression, *e.g.*, "`abc([^;,]*); (dd|ee)*;`". It can be a Prolog atom or string (*i.e.*, a list of characters). The above expression matches any substring that has "abc" followed by a sequence of characters none of which is a ";" or a ",", followed by a "; ", followed by a sequence that consists of zero or more of "dd" or "ee" segments, followed by a ";". An example of a string where such a match can be found is "`123abc&*^; ddeedd;poi`".

`InputStr` is the string to be matched against. It can be a Prolog atom or a string (list of characters). `Offset` is an integer offset into the string. The matching process starts at this offset. `IgnoreCase` indicates whether the case of the letters is to be ignored. If this argument is an uninstantiated variable, then the case is *not* ignored. If this argument is bound to an integer then the case *is* ignored.

The last argument, `MatchList`, is used to return the results. It must unify with a list of the form:

```
[match(beg_off0,end_off0), match(beg_off1,end_off1), ...]
```

The term `match(beg_off0,end_off0)` represents the substring that matches the *entire* regular expression, and the terms `match(beg_off1,end_off1)`, ..., represent the matches corresponding to the *parenthesized subexpressions* of the regular expression. The terms `beg_off` and `end_off` above are integers that specify beginning and ending offsets of the various matches. Thus, `beg_off0` is the offset into `InputStr` that points to the start of the maximal substring that matches the entire regular expression; `end_off0` points to the end of such a substring. In our case, the maximal matching substring is "`abc&*^; ddeedd;`" and the first term in the list returned by

```
| ?- re_match('abc([^;,]*); (dd|ee)*;', '123abc&*^; ddeedd;poi', 0, _,L).
```

is `match(3,18)`.

The most powerful feature of POSIX pattern matching is the ability to remember and return substrings matched by parenthesized subexpressions. When the above predicate succeeds, the terms 2,3, etc., in the above list represent the offsets for the matches corresponding to the parenthesized expressions 1,2,etc. For instance, our earlier regular expression

"`abc([^;,]*); (dd|ee)*;`" has two parenthetical subexpressions, which match "`&*^`" and "`dd`, respectively. So, the complete output from the above call is:

```
L = [match(3,18),match(6,9),match(15,17)]
```

The maximal number of parenthetical expressions supported by the Regmatch package is 30. Partial matches to parenthetical expressions 31 and over are discarded.

The match-terms corresponding to parenthetical expressions can sometimes report "*no-use*." This is possible when the regular expression specifies that zero or more occurrences of the parenthesized subexpression must be matched, and the match was made using zero subexpressions. In this case, the corresponding match term is `match(-1,-1)`. For instance,

```
| ?- re_match('ab(de)*', 'abcd',0,_,L).
L = [match(0,2),match(-1,-1)]
yes
```

Here the match that was found is the substring "ab" and the parenthesized subexpression "de" was not used. This fact is reported using the special match term `match(-1,-1)`.

Here is one more example of the power of POSIX regular expression matching:

```
| ?- re_match("a(b*|e*)cd\\1",'abbbcdbbbbbo', 0, _, M).
```

Here the result is:

```
M = [match(0,9),match(1,4)]
```

The interesting features here are the positional parameter \\1 and the alternating parenthetical expression `a(b*|e*)`. The alternating parenthetical expression here can match any sequence of b's *or* any sequence of e's. Note that if the string to be matched is not known when we write the program, we will not know a priori which sequence will be matched: a sequence of b's or a sequence of e's. Moreover, we do not even know the length of that sequence.

Now, suppose, we want to make sure that the matching substrings look like this:

```
abbbcdbbb
aeeeecdeeee
abbbbbbcdbbbbbb
```

How can we make sure that the suffix that follows "cd" is exactly the same string that is stuck between "a" and "cd"? This is what \\1 precisely does: it represents the substring matched by the first parenthetical expression. Similarly, you can use \\2, etc., if the regular expression contains more than one parenthetical expression.

The following example illustrates the use of the offset argument:

```
| ?- re_match("a(b*|e*)cd\\1",'abbbcdbbbbboabbbcdbbbbbo',2,_,M).

M = [match(12,21),match(13,16)]
```

Here, the string to be matched is double the string from the previous example. However, because we said that matching should start at offset 2, the first half of the string is not matched.

The `re_match/5` predicate fails if `Regexp` does not match `InputStr` or if the term specified in `MatchList` does not unify with the result produced by the match. Otherwise, it succeeds.

We should also note that parenthetical expressions can be represented using the `\(...\)` notation. What if you want to match a "(" then? You must escape it with a "\\" then:

```
| ?- re_match("a(b*)cd\\(",'abbbcd(bbo', 0, _, M).

M = [match(0,7),match(1,4)]
```

Now, what about matching the backslash itself? Try harder: you need four backslashes:

```
| ?- re_match("a(b*)cd\\\\",'abbbcd\bbo', 0, _, M).

M = [match(0,7),match(1,4)]
```

The predicate `re_bulkmatch/5` has the same calling sequence as `re_match/5`, and the meaning of the arguments is the same, except the last (output) argument. The difference is that `re_bulkmatch/5` ignores parenthesized subexpressions in the regular expression and instead of returning the matches corresponding to these parenthesized subexpressions it returns the list of all matches for the top-level regular expression. For instance,

```
| ?- re_bulkmatch('[^a-zA-Z0-9]+', '123&*-456 )7890% 123', 0, 1, X).

X = [match(3,6),match(9,11),match(15,17)]
```

Extracting the matches. The predicate `re_match/5` provides us with the offsets. How can we actually get the matched substrings? This is done with the help of the predicate `re_substring/4`:

```
re_substring(+String, +BeginOffset, +EndOffset, -Result).
```

This predicate works exactly like `substring/4` in XSB module `string` described in Part I of this manual.

Here is a complete example that shows matching followed by a subsequent extraction of the matches:

```
| ?- Str = 'abbbcd\bbo',
      re_match("a(b*)cd\\\\",Str,0,_,[match(X,Y), match(V,W)|L]),
      re_substring(Str,X,Y,Match),
      re_substring(Str,V,W,Paren1).

Str = abbbcd\bbo
X = 0
Y = 7
V = 1
W = 4
L = []
Match = abbbcd\
Paren1 = bbb
```

Substitution. The predicate `re_substitute/4` has the following invocation:

```
re_substitute(+InputStr, +SubstrList, +SubstitutionList, -OutStr)
```

This predicate works exactly like `string_substitute/4` in XSB module `string` described in Part I of this manual.

```
| ?- re_bulkmatch('[^a-zA-Z0-9]+', '123&*-456 )7890| 123', 0, _, X),
     re_substitute('123&*-456 )7890| 123', X, ['+++'], Y).

X = [match(3,6),match(9,11),match(15,17)]
Y = 123+++456+++7890+++123
```

7.2 `wildmatch`: Wildcard Matching and Globing

These interfaces are implemented using the `Wildmatch` package of XSB. This package provides the following functionality:

1. Telling whether a wildcard, like the ones used in Unix shells, match against a given string. Wildcards supported are of the kind available in tcsh or bash. Alternating characters (*e.g.*, "`[abc]`" or "`[^abc]`") are supported.

2. Finding the list of all file names in a given directory that match a given wildcard. This facility generalizes `directory/2` (in module `directory`), and it is much more efficient.

3. String conversion to lower and upper case.

To use this package, you need to type:

```
| ?- [wildmatch].
```

If you are planning to use it in an XSB program, you need this directive:

```
:- import glob_directory/4, wildmatch/3, convert_string/3 from wildmatch.
```

The calling sequence for `glob_directory/4` is:

```
glob_directory(+Wildcard, +Directory, ?MarkDirs, -FileList)
```

The parameter `Wildcard` can be either a Prolog atom or a Prolog string. `Directory` is also an atom or a string; it specifies the directory to be globbed. `MarkDirs` indicates whether directory names should be decorated with a trailing slash: if `MarkDirs` is bound, then directories will be so decorated. If MarkDirs is an unbound variable, then trailing slashes will not be added.

`FileList` gets the list of files in `Directory` that match `Wildcard`. If `Directory` is bound to an atom, then `FileList` gets bound to a list of atoms; if `Directory` is a Prolog string, then `FileList` will be bound to a list of strings as well.

This predicate succeeds is at least one match is found. If no matches are found or if `Directory` does not exist or cannot be read, then the predicate fails.

The calling sequence for `wildmatch/3` is as follows:

```
wildmatch(+Wildcard, +String, ?IgnoreCase)
```

`Wildcard` is the same as before. `String` represents the string to be matched against `Wildcard`. Like `Wildcard`, `String` can be an atom or a string. `IgnoreCase` indicates whether case of letters should be ignored during matching. Namely, if this argument is bound to a non-variable, then the case of letters is ignored. Otherwise, if `IgnoreCase` is a variable, then the case of letters is preserved.

This predicate succeeds when `Wildcard` matches `String` and fails otherwise.

The calling sequence for `convert_string/3` is as follows:

```
convert_string(+InputString, +OutputString, +ConversionFlag)
```

The input string must be an atom or a character list. The output string must be unbound. Its type will be "atom" if so was the input and it will be a character list if so was the input string. The conversion flag must be the atom `tolower` or `toupper`.

This predicate always succeeds, unless there was an error, such as wrong type argument passed as a parameter.

Chapter 8

curl: The XSB Internet Access Package

By Aneesh Ali

8.1 Introduction

The `curl` package is an interface to the `libcurl` library, which provides access to most of the standard Web protocols. The supported protocols include FTP, FTPS, HTTP, HTTPS, SCP, SFTP, TFTP, TELNET, DICT, LDAP, LDAPS, FILE, IMAP, SMTP, POP3 and RTSP. Libcurl supports SSL certificates, HTTP GET/POST/PUT/DELETE, FTP uploading, HTTP form based upload, proxies, cookies, user+password authentication (Basic, Digest, NTLM, Negotiate, Kerberos4), file transfer resume, HTTP proxy tunneling etc. The `curl` package of XSB supports a subset of that functionality, as described below.

The `curl` package accepts input in the form of URLs and Prolog atoms. To load the `curl` package, execute the following query in the XSB shell or loaded file:

```
?- [curl].
```

The `curl` package is integrated with file I/O of XSB in a transparent fashion and for many purposes Web pages can be treated just as yet another kind of a file. We first explain how Web pages can be accessed using the standard file I/O feature and then describe other predicates, which provide a lower-level interface.

8.2 Integration with File I/O

The `curl` package is integrated with XSB File I/O so that a web page can be opened as any other file.Once a Web page is opened, it can be read or written just like the a normal file.

8.2.1 Opening a Web Document

Web documents are opened by the usual predicates **see/1**, **open/3**, **open/4**.

see(*url*(*+Url*))

see(*url*(*+Url,Options*))

open(*url*(*+Url*), *+Mode*, *-Stream*)

open(*url*(*+Url*), *+Mode*, *-Stream*, *+Options*)

Url is an atom that specifies a URL. *Stream* is the file stream of the open file. *Mode* can be **read**, to create an input stream, or **write**, to create an output stream. For reading, the contents of the Web page are cached in a temporary file. For writing, a temporary empty file is created. This file is posted to the corresponding URL at closing.

The *Options* parameter is a list that controls loading. Members of that list can be of the following form:

redirect(*Bool*)
: Specifies the redirection option. The supported values are true and false. If true, any number of redirects is allowed. If false, redirections are ignored. The default is true.

secure(*CrtName*)
: Specifies the secure connections (https) option. *CrtName* is the name of the file holding one or more certificates to verify the peer with.

auth(*UserName, Password*)
: Sets the username and password basic authentication.

timeout(*Seconds*)
: Sets the maximum time in seconds that is allowed for the transfer operation.

user_agent(*Agent*)
: Sets the User-Agent: header in the http request sent to the remote server.

header(*String*)
: This allows one to specify an HTTP header. Several `header(...)` options can be specified in the same list. Specifying the headers is useful mostly when closing Web pages that are open for writing, which corresponds to POST HTTP requests.

8.2.2 Closing a Web Document

Web documents opened by the predicates **see/1**, **open/3**, and **open/4** above must be closed by the predicates **close/2** or **close/4**. The stream corresponding to the URL is closed. If the stream was open for writing, the data written to the stream is POSTed to the URL, which corresponds to HTTP POST. If writing is unsuccessful for some reason, a list of warnings is returned.

close(*+Source, +Options*)

close(*+Source, +Options, -Response, -Warnings*)
These versions of `close` are typically used for sources that are open for writing. URL-streams open for reading can be closed using the usual `close/1` predicate. *Source* is of the form `url(`*url-string*`)`, where *url-string* must be an atom. *Options* is a list of options like those for the open predicate of Section 8.2.1. If the HTTP server returns a response, the *Response* variable is bound to that string. *Warnings* is a list of possible warnings. If everything is fine, this list is empty. Closing often requires the `header(...)` option because it is often necessary to specify `Content-Type` and other header attributes when posting to a Web site.

8.3 Low Level Predicates

This section describes additional predicates provided by the `curl` packages, which extend the functionality provided by the file I/O integration.

8.3.1 Loading Web Documents

Web documents are loaded by the predicate **load_page/5**, which has many options. The parameters of this predicate are described below.

load_page(*+Source, +Options, -Properties, -Content, -Warn*)
Source is of the form `url(`*url*`)`. The document is returned in *Content*. *Warn* is bound to a (possibly empty) list of warnings generated during the process.

Properties is bound to a *list* of properties of the document. They are *Page size*, *Page last modification time*, and *Redirection URL*. The `load_page/5` predicate caches a copy of the Web page that it fetched from the Web in a local file, which is identified by the URL's directory, file name portion, and its file extension. The first two parameters indicate the size and the last modification time of the fetched Web page. The last parameter, *Redirection URL*, is the source URL, if no redirection happened or, if the

original URL was redirected then this parameter shows the final URL. The directory and the file name The *Options* parameter is the same as in Section 8.2.1.

`load_page` has additional options that can appear in the *Options* list:

`post_data`(*String*)
: This allows one to post data (HTTP POST) to a web page and is an alternative to posting by opening-writing-closing URLs, which was described above.

  If several `post_data(...)` options are given, only the last one is used. This option often goes with the `header(...)` option because it is often necessary to specify `Content-Type` and other header attributes.

  If this option is specified with an `open/4` predicate, it is ignored. If it is specified in the `close/2` or `close/4` predicate, *String* is posted instead of what was written to the closed stream. This goes to say that the `post_data` option in `open` and `close` predicates makes little sense.

`put_data`(*String*)
: This allows one to put data (HTTP PUT) to a web page.

  If several `put_data(...)` options are given, only the last one is used. This option often goes with the `header(...)` option because it is often necessary to specify `Content-Type` and other header attributes.

  If this option is specified with an `open/4`, `close/2`, or `close/4` predicate, it is ignored.

`delete`
: This sends a DELETE HTTP request to the server.

8.3.2 Retrieving Properties of a Web Document

The properties of a web document are loaded by the predicates **url_properties/3** and **url_properties/2**.

url_properties(*+Url, +Options, -Properties*)
: The *Options* and *Properties* are same as in **load_page/5**: a *list* of properties of the document, which are *Page size*, *Page last modification time*, *RedirectionURL* in that order. If the original page has no redirection then *RedirectionURL* is the same as *Url*. Some Web servers will not report page sizes or modification times (or both) in which case they appear as -1.

url_properties(*+Url, -Properties*)
: This uses the default options (`secure(false)`, `redirect(true)`).

8.3.3 Encoding URLs

Sometimes it is necessary to convert a URL string into something that can be used, for example, as a file name. This is done by the following predicate.

encode_url(*+Source, -Result***)**

Source has the form *url(url-string)*, where *url-string* is an atom. *Result* is bound to a list of components of the URL: the URL-encoded *Directory Name*, the URL-encoded *File Name*, and the *Extension* of the URL.

8.4 Installation and configuration

The `curl` package of XSB requires that the `libcurl` package is installed. For Windows, the `libcurl` library files are included with the installation. For Linux and Mac, the `libcurl` and `libcurl-dev` packages need to be installed using a suitable package manager (e.g., deb or rpm in Linux, Homebrew in Mac). In some systems, `libcurl-dev` might be called `libcurl-gnutls-dev` or `libcurl-openssl-dev`. In addition, the release number might be attached, as in `libcurl4` and `libcurl4-openssl-dev`.

The `libcurl` package can also be downloaded and built manually from

http://curl.haxx.se/download.html

To configure `curl` on Linux, Mac, or on some other Unix variant, switch to the `XSB/build` directory and type

```
cd XSB/packages/curl
./configure
./makexsb
```

Chapter 9

Packages sgml and xpath: SGML/XML/HTML and XPath Parsers

By Rohan Shirwaikar

9.1 Introduction

This suite of packages consists of the `sgml` package, which can parse XML, HTML, XHTML, and even SGML documents and the `xpath` package, which supports XPath queries on XML documents. The `sgml` package is an adaptation of a similar package in SWI Prolog and a port of SWI's codebase with some minor changes. The `xpath` package provides an interface to the popular `libxml2` library, which supports XPath and XML parsing, and is used in Mozilla based browsers. At present, the XML parsing capabilities of `libxml2` are not utilized explicitly in XSB, but such support might be provided in the future. The `sgml` package does not rely on `libxml2` [1].

Installation and configuration. The `sgml` package does not require any installation steps under Unix-based systems or under Cygwin. Under native Windows, if you downloaded XSB from SVN, you need to compile the package as follows:

```
cd XSB\packages\sgml\cc
nmake /f NMakefile.mak
```

[1] This package has not yet been tested for thread-safety

You need MS Visual Studio for that. If you downloaded a prebuilt version of XSB, then the `sgml` package should have already been compiled for you and no installation is required.

The details of the `xpath` package and the corresponding configuration instructions appear in Section 9.4.

9.2 Overview of the SGML Parser

The `sgml` package accepts input in the form of files, URLs and Prolog atoms. To load the `sgml` parser, the user should type

```
?- [sgml].
```

at the prompt. If `test.html` is a file with the following contents

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">

<html>
<head>
<title>Demo</title>
</head>
<body>

<h1 align=center>This is a demo</h1>

<p>Paragraphs in HTML need not be closed.

<p>This is called ‘omitted-tag’ handling.
</body>
</html>
```

then the following call

```
?- load_html_structure(file('test.html'), Term, Warn).
```

will parse the document and bind `Term` to the following Prolog term:

```
[ element(html,
          [],
          [ element(head,
                    [],
```

```
                    [ element(title,
                              [],
                              [ 'Demo'
                              ])
                    ]),
            element(body,
                    [],
                    [ '\n',
                      element(h1,
                              [ align = center
                              ],
                              [ 'This is a demo'
                              ]),
                      '\n\n',
                      element(p,
                              [],
                              [ 'Paragraphs in HTML need not be closed.\n'
                              ]),
                      element(p,
                              [],
                              [ 'This is called `omitted-tag\' handling.'
                              ])
                    ])
          ])
].
```

The XML document is converted into a list of Prolog terms of the form

`element(`*`Name`*`,`*`Attributes`*`,`*`Content`*`)`.

Each term corresponds to an XML element. *Name* represents the name of the element. *Attributes* is a list of attribute-value pairs of the element. *Content* is a list of child-elements and CDATA (general character data). For instance,

```
<aaa>fooo<bbb>foo1</bbb></aaa>
```

will be parsed as

```
element(aaa,[],[fooo, element(bbb,[],[foo1])])
```

Entities (e.g. `<`) are returned as part of `CDATA`, unless they cannot be represented. Each entity is clothed in the term `entity/1`. See **load_sgml_structure/3** for details.

9.3 Predicate Reference

9.3.1 Loading Structured Documents

SGML, HTML, and XML documents are parsed by the predicate **load_structure/4**, which has many options. For convenience, a number of commonly used shorthands are provided to parse SGML, XML, HTML, and XHTML documents respectively.

load_sgml_structure(*+Source, -Content, -Warn***)**

load_xml_structure(*+Source, -Content, -Warn***)**

load_html_structure(*+Source, -Content, -Warn***)**

load_xhtml_structure(*+Source, -Content, -Warn***)**

The parameters of these predicates have the same meaning as those in **load_structure/4**, and are described below.

The above predicates (in fact, just `load_xml_structure/3` and `load_html_structure/3`) are the most commonly used predicates of the `sgml` package. The other predicates described in this section are needed only for advanced uses of the package.

load_structure(*+Source, -Content, +Options, -Warn***)**

Source can have one of the following forms: `url(`*url*`)`, `file(`*file name*`)`, `string(`*'document as a Prolog atom'*`)`. The parsed document is returned in *Content*. *Warn* is bound to a (possibly empty) list of warnings generated during the parsing process. *Options* is a list of parameters that control parsing, which are described later.

The list *Content* can have the following members:

A Prolog atom

Atoms are used to represent character strings, i.e., `CDATA`.

element(*Name, Attributes, Content***)**

Name is the name of the element tag. Since SGML is case-insensitive, all element names are returned as lowercase atoms.

Attributes is a list of pairs the form *Name*=*Value*, where *Name* is the name of an attribute and *Value* is its value. Values of type `CDATA` are represented as atoms. The values of multi-valued attributes (`NAMES`, *etc.*) are represented as a lists of atoms. Handling of the attributes of types `NUMBER` and `NUMBERS` depends on the setting of the `number(+NumberMode)` option of **set_sgml_parser/2** or **load_structure/3** (see later). By default the values of such attributes are represented as atoms, but the `number(...)` option can also specify that these values must be converted to Prolog integers.

Content is a list that represents the content for the element.

entity(*Code*)
: If a character entity (e.g., `&#913;`) is encountered that cannot be represented in the Prolog character set, this term is returned. It represents the code of the encountered character (e.g., `entity(913)`).

entity(*Name*)
: This is a special case of `entity(Code)`, intended to handle special symbols by their name rather than character code. If an entity refers to a character entity holding a single character, but this character cannot be represented in the Prolog character set, this term is returned. For example, if the contents of an element is `&Alpha; < &Beta;` then it will be represented as follows:

```
[ entity('Alpha'), ' < ', entity('Beta') ]
```

Note that entity names are case sensitive in both SGML and XML.

sdata(*Text*)
: If an entity with declared content-type `SDATA` is encountered, this term is used. The data of the entity instantiates *Text*.

ndata(*Text*)
: If an entity with declared content-type `NDATA` is encountered, this term is used. The data instantiates *Text*.

pi(*Text*)
: If a processing instruction is encountered (`<?...?>`), *Text* holds the text of the processing instruction. Please note that the `<?xml ...?>` instruction is ignored and is not treated as a processing instruction.

The *Options* parameter is a list that controls parsing. Members of that list can be of the following form:

dtd(*?DTD*)
: Reference to a DTD object. If specified, the `<!DOCTYPE ...>` declaration supplied with the document is ignored and the document is parsed and validated against the provided DTD. If the DTD argument is a variable, then a the variable *DTD* gets bound to the DTD object created out of the DTD supplied with the document.

dialect(*+Dialect*)
: Specify the parsing dialect. The supported dialects are `sgml` (default), `xml` and `xmlns`.

space(*+SpaceMode*)
: Sets the space handling mode for the initial environment. This mode is inherited by the other environments, which can override the inherited value using the XML reserved attribute **xml:space**. See Section 9.3.2 for details.

number(*+NumberMode*)
: Determines how attributes of type `NUMBER` and `NUMBERS` are handled. If `token`

is specified (the default) they are passed as an atom. If `integer` is specified the parser attempts to convert the value to an integer. If conversion is successful, the attribute is represented as a Prolog integer. Otherwise the value is represented as an atom. Note that SGML defines a numeric attribute to be a sequence of digits. The `-` (minus) sign is not allowed and `1` is different from `01`. For this reason the default is to handle numeric attributes as tokens. If conversion to integer is enabled, negative values are silently accepted and the minus sign is ignored.

defaults(*+Bool***)**
Determines how default and fixed attributes from the DTD are used. By default, defaults are included in the output if they do not appear in the source. If `false`, only the attributes occurring in the source are emitted.

file(*+Name***)**
Sets the name of the input file for error reporting. This is useful if the input is a stream that is not coming from a file. In this case, errors and warnings will not have the file name in them, and this option allows one to force inclusion of a file name in such messages.

line(*+Line***)**
Sets the starting line-number for reporting errors. For instance, if `line(10)` is specified and an error is found at line X then the error message will say that the error occurred at line X+10. This option is used when the input stream does not start with the first line of a file.

max_errors(*+Max***)**
Sets the maximum number of errors. The default is 50. If this number is reached, the following exception is raised:

```
error(limit_exceeded(max_errors, Max), _)
```

9.3.2 Handling of White Spaces

Four modes for handling white-spaces are provided. The initial mode can be switched using the `space(SpaceMode)` option to **load_structure/3** or **set_sgml_parser/2**. In XML mode, the mode is further controlled by the **xml:space** attribute, which may be specified both in the DTD and in the document. The defined modes are:

space(sgml)
Newlines at the start and end of an element are removed. This is the default mode for the SGML dialect.

space(preserve)
White space is passed literally to the application. This mode leaves all white space handling to the application. This is the default mode for the XML dialect.

space(default)
In addition to `sgml` space-mode, all consecutive whitespace is reduced to a single space-character.

space(remove)
In addition to `default`, all leading and trailing white-space is removed from `CDATA` objects. If, as a result, the `CDATA` becomes empty, nothing is passed to the application. This mode is especially handy for processing data-oriented documents, such as RDF. It is not suitable for normal text documents. Consider the HTML fragment below. When processed in this mode, the spaces surrounding the three elements in the example below are lost. This mode is not part of any standard: XML 1.0 allows only `default` and `preserve`.

```
Consider adjacent <b>bold</b> <ul>and</ul> <it>italic</it> words.
```

The parsed term will be `['Consider adjacent',element(b,[],[bold]),element(ul,[],[and]),element(it,[],[italics]),words]`.

9.3.3 XML documents

The parser can operate in two modes: the `sgml` mode and the `xml` mode, as defined by the `dialect(Dialect)` option. HTML is a special case of the SGML mode with a particular DTD. Regardless of this option, if the first line of the document reads as below, the parser is switched automatically to the XML mode.

```
<?xml ... ?>
```

Switching to XML mode implies:

- *XML empty elements*
  The construct `<element attribute ... attribute/>` is recognized as an empty element.
- *Predefined entities*
  The following entities are predefined: `&lt;` (<), `&gt;` (>), `&amp;` (&), `&apos;` (') and `&quot;` (").
- *Case sensitivity*
  In XML mode, names of tags and attributes are case-sensitive, except for the DTD reserved names (i.e. `ELEMENT`, *etc.*).
- *Character classes*
  In XML mode, underscore (_) and colon (:) are allowed in names.

- *White-space handling*
  White space mode is set to `preserve`. In addition, the XML reserved attribute **xml:space** is honored; it may appear both in the document and the DTD. The `remove` extension (see `space(remove)` earlier) is allowed as a value of the **xml:space** attribute. For example, the DTD statement below ensures that the **pre** element preserves space, regardless of the default processing mode.

  ```
  <!ATTLIST pre xml:space nmtoken #fixed preserve>
  ```

XML Namespaces

Using the dialect `xmlns`, the parser will recognize XML namespace prefixes. In this case, the names of elements are returned as a term of the format

URL:*LocalName*

If an identifier has no namespace prefix and there is no default namespace, it is returned as a simple atom. If an identifier has a namespace prefix but this prefix is undeclared, the namespace prefix rather than the related URL is returned.

Attributes declaring namespaces (`xmlns:`*ns*=*url*) are represented in the translation as regular attributes.

9.3.4 DTD-Handling

The DTD (**D**ocument **T**ype **D**efinition) are internally represented as objects that can be created, freed, defined, and inspected. Like the parser itself, it is filled by opening it as a Prolog output stream and sending data to it. This section summarizes the predicates for handling the DTD.

new_dtd(*+DocType, -DTD, -Warn*)
: Creates an empty DTD for the named *DocType*. The returned DTD-reference is an opaque term that can be used in the other predicates of this package. *Warn* is the list of warnings generated.

free_dtd(*+DTD, -Warn*)
: Deallocate all resources associated to the DTD. Further use of *DTD* is invalid. *Warn* is the list of warnings generated.

open_dtd(*+DTD, +Options, -Warn*)
: This opens and loads a DTD from a specified location (given in the *Options* parameter

(see next). *DTD* represents the created DTD object after the source is loaded. *Options* is a list options. Currently the only option supported is *source(location)*, where *location* can be of one of these forms:

`url(`*url*`)`
`file(`*fileName*`)`
`string(`*'document as a Prolog atom'*`)`.

dtd(*+DocType, -DTD, -Warn***)**
Certain DTDs are part of the system and have known doctypes. Currently, 'HTML' and 'XHTML' are the only recognized built-in doctypes. Such a DTD can be used for parsing simply by specifying the doctype. Thus, the `dtd/3` predicate takes the doctype name, finds the DTD associated with the given doctype, and creates a dtd object for it. *Warn* is the list of warnings generated.

dtd(*+DocType, -DTD, +DtdFile -Warn***)**

The predicate parses the DTD present at the location *DtdFile* and creates the corresponding DTD object. *DtdFile* can have one of the following forms: `url(`*url*`)`, `file(`*fileName*`)`, `string(`*'document as a Prolog atom'*`)`.

9.3.5 Low-level Parsing Primitives

The following primitives are used only for more complex types of parsing, which might not be covered by the `load_structure/4` predicate.

new_sgml_parser(*-Parser, +Options, -Warn***)**
Creates a new parser. *Warn* is the list of warnings generated. A parser can be used one or multiple times for parsing documents or parts thereof. It may be bound to a DTD or the DTD may be left implicit. In this case the DTD is created from the document prologue or (if it is not in the prologue) parsing is performed without a DTD. The *Options* list can contain the following parameters:

dtd(*?DTD***)**
If *DTD* is bound to a DTD object, this DTD is used for parsing the document and the document's prologue is ignored. If *DTD* is a variable, the variable gets bound to a created DTD. This DTD may be created from the document prologue or build implicitly from the document's content.

free_sgml_parser(*+Parser, -Warn***)**
Destroy all resources related to the parser. This does not destroy the DTD if the parser was created using the `dtd(`*DTD*`)` option. *Warn* is the list of warnings generated during parsing (can be empty).

set_sgml_parser(*+Parser, +Option, -Warn*)
Sets attributes to the parser. *Warn* is the list of warnings generated. *Options* is a list that can contain the following members:

file(*File*)
Sets the file for reporting errors and warnings. Sets the linenumber to 1.

line(*Line*)
Sets the starting line for error reporting. Useful if the stream is not at the start of the (file) object for generating proper line-numbers. This option has the same meaning as in the `load_structure/4` predicate.

charpos(*Offset*)
Sets the starting character location. See also the `file(File)` option. Used when the stream does not start from the beginning of a document.

dialect(*Dialect*)
Set the markup dialect. Known dialects:

`sgml`
The default dialect. This implies markup is case-insensitive and standard SGML abbreviation is allowed (abbreviated attributes and omitted tags).

`xml`
This dialect is selected automatically if the processing instruction `<?xml ...>` is encountered.

`xmlns`
Process file as XML file with namespace support.

qualify_attributes(*Boolean*)
Specifies how to handle unqualified attributes (i.e., without an explicit namespace) in XML namespace (`xmlns`) dialect. By default, such attributes are not qualified with namespace prefixes. If `true`, such attributes are qualified with the namespace of the element they appear in.

space(*SpaceMode*)
Define the initial handling of white-space in `PCDATA`. This attribute is described in Section 9.3.2.

number(*NumberMode*)
If `token` is specified (the default), attributes of type number are represented as a Prolog atom. If `integer` is specified, such attributes are translated into Prolog integers. If the conversion fails (e.g., due to an overflow) a warning is issued and the value is represented as an atom.

doctype(*Element*)
Defines the top-level element of the document. If a `<!DOCTYPE ...>` declaration has been parsed, this declaration is used. If there is no `DOCTYPE` declaration then

the parser can be instructed to use the element given in `doctype(_)` as the top level element. This feature is useful when parsing part of a document (see the `parse` option to **sgml_parse/3**).

sgml_parse(*+Parser, +Options, -Warn***)**

Parse an XML file. The parser can operate in two input and two output modes. Output is a structured term as described with **load_structure/4**.

Warn is the list of warnings generated. A full description of *Options* is given below.

document(*+Term***)**

A variable that will be unified with a list describing the content of the document (see **load_structure/4**).

source(*+Source***)**

Source can have one of the following forms: `url(`*url*`)`, `file(`*fileName*`)`, `string(`*'document as a Prolog atom'*`)`. This option *must* be given.

content_length(*+Characters***)**

Stop parsing after the given number of *Characters*. This option is useful for parsing input embedded in *envelopes*, such as HTTP envelopes.

parse(*Unit***)**

Defines how much of the input is parsed. This option is used to parse only parts of a file.

`file`

Default. Parse everything upto the end of the input.

`element`

The parser stops after reading the first element. Using `source(`*`Stream`*`)`, this implies reading is stopped as soon as the element is complete, and another call may be issued on the same stream to read the next element.

`declaration`

This may be used to stop the parser after reading the first declaration. This is useful if we want to parse only the `doctype` declaration.

max_errors(*+MaxErrors***)**

Sets the maximum number of errors. If this number is exceeded, further writes to the stream will yield an I/O error exception. Printing of errors is suppressed after reaching this value. The default is 100.

syntax_errors(*+ErrorMode***)**

Defines how syntax errors are handled.

quiet

Suppress all messages.

print

Default. Print messages.

9.3.6 External Entities

While processing an SGML document the document may refer to external data. This occurs in three places: external parameter entities, normal external entities and the `DOCTYPE` declaration. The current version of this tool deals rather primitively with external data. External entities can only be loaded from a file.

Two types of lines are recognized by this package:

`DOCTYPE` *doctype* *file*

`PUBLIC "Id "` *file*

The parser loads the entity from the file specified as *file*. The file can be local or a URL.

9.3.7 Exceptions

Exceptions are generated by the parser in two cases. The first case is when the user specifies wrong input. For example when specifying

```
load_structure( string('<m></m>'), Document, [line(xyz)], Warn)
```

The string `xyz` is not in the domain of `line`. Hence in this case a domain error exception will be thrown.

Exceptions are generated when XML being parsed is not well formed. For example if the input XML contains

```
'<m></m1>'
```

exceptions will be thrown.

In both cases the format of the exception is

`error( sgml(` *error term*`),` *error message*`)`
`warning( sgml(` *warning term*`),` *warning message*`)`

where *error term* or *warning term* can be of the form

- *pointer to the parser instance*,
- *line at which error occurred*,

- *error code.*
- *functor(argument)*, where *functor* and *argument* depend on the type of exception raised. For example,

  `resource-error(no-memory)` — if memory is unavailable

  `permission-error(file-name)` — no permission to read a file

  A `system-error(description) -- internal system error`

  `type-error(expected,actual)` — data type error

  `domain-error(functor,offending-value)` — the offending value is not in the domain of the functor. For instance, in `load_structure( string('<m></m>'), Document, [line(xyz)], Warn)`, `xyz` is not in the domain of `line`.

  `existence-error(resource)` — resource does not exist

  `limit-exceeded(limit,maxval)` — value exceeds the limit.

9.3.8 Unsupported features

The current parser is rather limited. While it is able to deal with many serious documents, it omits several less-used features of SGML and XML. Known missing SGML features include

- *NOTATION on entities*
  Though notation is parsed, notation attributes on external entity declarations are not represented in the output.
- *NOTATION attributes*
  SGML notations may have attributes, declared using `<!ATTLIST #NOT name attrib>`. Those data attributes are provided when you declare an external CDATA, NDATA, or SDATA entity. XML does not support external CDATA, NDATA, or SDATA entities, nor any of the other uses to which data attributes are put in SGML.
- *SGML declaration*
  The 'SGML declaration' is fixed, though most of the parameters are handled through indirections in the implementation.
- *The RANK feature*
  It is regarded as obsolete.
- *The LINK feature*
  It is regarded as too complicated.
- *The CONCUR feature*
  Concurrent markup allows a document to be tagged according to more than one DTD at the same time. It is not supported.

- *The Catalog files*
  Catalog files are not supported.

In the XML mode, the parser recognizes SGML constructs that are not allowed in XML. Also various extensions of XML over SGML are not yet realized. In particular, XInclude is not implemented.

9.3.9 Summary of Predicates

dtd/2	Find or build a DTD for a document type
free_dtd/1	Free a DTD object
free_sgml_parser/1	Destroy a parser
load_dtd/2	Read DTD information from a file
load_structure/4	Parse XML/SGML/HTML data into Prolog term
load_sgml_structure/3	Parse SGML file into Prolog term
load_html_structure/3	Parse HTML file into Prolog term
load_xml_structure/3	Parse XML file into Prolog term
load_xhtml_structure/3	Parse XHTML file into Prolog term
new_dtd/2	Create a DTD object
new_sgml_parser/2	Create a new parser
open_dtd/3	Open a DTD object as an output stream
set_sgml_parser/2	Set parser options (dialect, source, *etc.*)
sgml_parse/2	Parse the input
xml_name/1	Test atom for valid XML name
xml_quote_attribute/2	Quote text for use as an attribute
xml_quote_cdata/2	Quote text for use as PCDATA

9.4 XPath support

XPath is a query language for addressing parts of an XML document. In XSB, this support is provided by the `xpath` package. To use this package the `libxml2` XML parsing library must be installed on the machine. It comes with most Linux distributions, since it is part of the Gnome desktop, or one can download it from *http://xmlsoft.org/*. It is available for Linux, Solaris, Windows, and MacOS. Note that both the library itself and the `.h` files of that library must be installed. In some Linux distributions, the `.h` files might reside in a separate package from the package that contains the actual library. For instance, the library (`libxml2.so`) might be in the package called `libxml2` (which is usually installed by default), while the `.h` files might be in the package `libxml2-dev` (which is usually *not* in default installations).

On Unix-based systems (and MacOS), the package might need to be configured at the

time XSB is configured using XSB's `configure` script found in the XSB's `build` directory. Normally, if `libxml2` is installed by a Linux package manager, nothing special is required: the package will be configured by default. If the library is in a non-standard place, then the configure option `-with-xpath-dir=`*directory-of-libxml2* must be given. It must specify the directory where `lib/*/libxml2.so` (or libxml2.dylib in Mac) and `include/libxml2` can be found.

Examples: If libxml2 is in a default location, then XSB can be configured simply like this:

```
./configure
```

Otherwise, use

```
./configure --with-xpath-dir=/usr/local
```

if, for example, `libxml2.so` is in `/usr/local/lib/i386-linux-gnu/libxml2.so` and the included `.h` files are in `/usr/local/include/libxml2/*`.

On Windows and under Cygwin, the `libxml2` library is already included in the XSB distribution and does not need to be downloaded. If you are using a prebuilt XSB distribution for Windows, then you do not need to do anything—the package has already been built for you.

For Cygwin, you only need to run the `./configure` script without any options. This needs to be done regardless of whether you downloaded XSB from CVS or a released prebuilt version.

If you downloaded XSB from CVS and want to use it under native Windows (not Cygwin), then you would need to compile the XPath package, and you need Microsoft's Visual Studio. To compile the package one should do the following:

```
cd packages\xpath\cc
nmake /f NMakefile.mak
```

The following section assumes that the reader is familiar with the syntax of XPath and its capabilities. To load the `xpath` package, type

```
:-[xpath].
```

The program needs to include the following directive:

```
:- import parse_xpath/4 from xpath.
```

XPath query evaluation is done by using the `parse_xpath` predicate.

parse_xpath(*+Source, +XPathQuery, -Output, +NamespacePrefixList***)**

Source is a term of the format `url(`*url*`)`, `file(`*filename*`)` or `string(`*'XML-document-as-a-string'*`)`. It specifies that the input XML document is contained in a file, can be fetched from a URL, or is given directly as a Prolog atom.

XPathQuery is a standard XPath query which is to be evaluated on the XML document in *Source*.

Output gets bound to the output term. It represents the XML element returned after the XPath query is evaluated on the XML document in *Source*. The output term is of the form `string(`*'XML-document'*`)`. It can then be parsed using the `sgml` package described earlier.

NamespacePrefixList is a space separated list of pairs of the form *prefix* = *namespace*. This specifies the namespace prefixes that are used in the XPath query.

For example if the xpath expression is `'/x:html/x:head/x:meta'` where `x` is a prefix that stands for `'http://www.w3.org/1999/xhtml'`, then `x` would have to be defined as follows:

```
?- parse_xpath(url('http://w3.org'), '/x:html/x:head/x:meta', O4,
                    'x=http://www.w3.org/1999/xhtml').
```

In the above, the xpath query is `'/x:html/x:head/x:meta'` and the prefix has been defined as `'x=http://www.w3.org/1999/xhtml'`.

Chapter 10

rdf: The XSB RDF Parser

By Aneesh Ali

10.1 Introduction

RDF is a W3C standard for representing meta-data about documents on the Web as well as exchanging frame-based data (e.g. ontologies). RDF has a formal data model defined in terms of *triples*. In addition, a *graph* model is defined for visualization and an XML serialization for exchange. This chapter describes the API provided by the XSB RDF parsing package. The package and its documentation are adaptations from SWI Prolog.

Note that this package only handles RDF-XML. For RDF in Turtle, ntriples, etc. use xsbpy together with the Python rdflib package as indicated in Chapter 17.

10.2 High-level API

The RDF translator is built in Prolog on top of the **sgml2pl** package, which provides XML parsing. The transformation is realized in two passes. It is designed to operate in various environments and therefore provides interfaces at various levels. First we describe the top level, which parses RDF-XML file into a list of triples. These triples are *not* asserted into the Prolog database because it is not necessarily the final format the user wishes to use and it is not clear how the user might want to deal with multiple RDF documents. Some options are using global URI's in one pool, in Prolog modules, or using an additional argument.

load_rdf(*+File, -Triples***)**
Same as `load_rdf(`*+File, -Triples, []*).

load_rdf(*+File, -Triples, +Options*)

Read the RDF-XML file *File* and return a list of *Triples*. *Options* is a list of additional processing options. Currently defined options are:

base_uri(*BaseURI*)

If provided, local identifiers and identifier-references are globalized using this URI. If omitted, local identifiers are not tagged.

blank_nodes(*Mode*)

If *Mode* is `share` (default), blank-node properties (i.e. complex properties without identifier) are reused if they result in exactly the same triple-set. Two descriptions are shared if their intermediate description is the same. This means they should produce the same set of triples in the same order. The value `noshare` creates a new resource for each blank node.

expand_foreach(*Boolean*)

If *Boolean* is `true`, expand `rdf:aboutEach` into a set of triples. By default the parser generates `rdf`(`each`(*Container*), *Predicate*, *Subject*).

lang(*Lang*)

Define the initial language (i.e. pretend there is an `xml:lang` declaration in an enclosing element).

ignore_lang(*Bool*)

If `true`, `xml:lang` declarations in the document are ignored. This is mostly for compatibility with older versions of this library that did not support language identifiers.

convert_typed_literal(*:ConvertPred*)

If the parser finds a literal with the `rdf:datatype`=*Type* attribute, call *ConvertPred*(*+Type, +Content, -Literal*). *Content* is the XML element contents returned by the XML parser (a list). The predicate must unify *Literal* with a Prolog representation of *Content* according to *Type* or throw an exception if the conversion cannot be made.

This option serves two purposes. First of all it can be used to ignore type declarations for backward compatibility of this library. Second it can be used to convert typed literals to a meaningful Prolog representation (e.g., convert '42' to the Prolog integer 42 if the type is `xsd:int` or a related type).

namespaces(*-List*)

Unify *List* with a list of *NS*=*URL* for each encountered `xmlns`:*NS*=*URL* declaration found in the source.

entity(*+Name, +Value*)

Overrule entity declaration in file. As it is common practice to declare namespaces using entities in RDF/XML, this option allows changing the namespace without changing the file. Multiple such options are allowed.

The *Triples* list is a list of the form `rdf`(*Subject, Predicate, Object*) triples. *Subject* is either a plain resource (an atom), or one of the terms `each`(*URI*) or `prefix`(*URI*) with the usual meaning. *Predicate* is either a plain atom for explicitly non-qualified names or a term *NameSpace*:*Name*. If *NameSpace* is the defined RDF name space it is returned as the atom `rdf`. *Object* is a URI, a *Predicate* or a term of the form `literal`(*Value*) for literal values. *Value* is either a plain atom or a parsed XML term (list of atoms and elements).

10.2.1 RDF Object representation

The *Object* (3rd) part of a triple can have several different types. If the object is a resource it is returned as either a plain atom or a term *NameSpace*:*Name*. If it is a literal it is returned as `literal`(*Value*), where *Value* can have one of the form below.

- An atom
  If the literal *Value* is a plain atom is a literal value not subject to a datatype or `xml:lang` qualifier.
- **lang**(*LanguageID, Atom*)
  If the literal is subject to an `xml:lang` qualifier *LanguageID* specifies the language and *Atom* the actual text.
- A list
  If the literal is an XML literal as created by `parseType="Literal"`, the raw output of the XML parser for the content of the element is returned. This content is a list of `element`(*Name, Attributes, Content*) and atoms for CDATA parts as described with the `sgml` package.
- **type**(*Type, StringValue*)
  If the literal has an `rdf:datatype`=*Type* a term of this format is returned.

10.2.2 Name spaces

RDF name spaces are identified using URIs. Unfortunately various URI's are in common use to refer to RDF. The RDF parser therefore defines the `rdf_name_space/1` predicate as `multifile`, which can be extended by the user. For example, to parse Netscape OpenDirectory (http://www.mozilla.org/rdf/doc/inference.html) given in the `structure.rdf` file (http://rdf.dmoz.org/rdf/structure.rdf.u8.gz), the following declarations are used:

```
:- multifile
        rdf_parser:rdf_name_space/1.
```

```
rdf_parser:rdf_name_space('http://www.w3.org/TR/RDF/').
rdf_parser:rdf_name_space('http://directory.mozilla.org/rdf').
rdf_parser:rdf_name_space('http://dmoz.org/rdf').
```

The above statements will then extend the initial definition of this predicate provided by the parser:

```
rdf_name_space('http://www.w3.org/1999/02/22-rdf-syntax-ns#').
rdf_name_space('http://www.w3.org/TR/REC-rdf-syntax').
```

10.2.3 Low-level access

The predicates `load_rdf/2` and `load_rdf/3` described earlier are not always sufficient. For example, they cannot deal with documents where the RDF statement is embedded in an XML document. It also cannot deal with really large documents (e.g. the Netscape OpenDirectory project, currently about 90 MBytes), without requiring huge amounts of memory.

For really large documents, the **sgml2pl** parser can be instructed to handle the content of a specific element (i.e. `<rdf:RDF>`) element-by-element. The parsing primitives defined in this section can be used to process these one-by-one.

xml_to_rdf(*+XML, +BaseURI, -Triples*)
: Process an XML term produced by `sgml`'s `load_structure/4` using the `dialect(xmlns)` output option. *XML* is either a complete `<rdf:RDF>` element, a list of RDF-objects (container or description), or a single description of container.

10.3 Testing the RDF translator

A test-suite and a driver program are provided by `rdf_test.P` in the `XSB/examples/rdf` directory. To run these tests, load this file into Prolog and execute `test_all`. The test files found in the directory `examples/rdf/suite` are then converted into triples. The expected output is in `examples/rdf/expectedoutput`. One can also run the tests selectively, using the following predicates:

suite(*+N*)
: Run test *N* using the file `suite/tN.rdf` and display its RDF representation and the triples.

test_file(*+File*)
: Process `File` and display its RDF representation and the triples.

Chapter 11

Constraint Packages

Constraint packages are an important part of modern logic programming, but approaches to constraints differ both in their semantics and in their implementation. At a semantic level, *Constraint Logic Programming* associates constraints with logical variables, and attempts to determine solutions that are inconsistent with or entailed by those constaints. At an implementational level, the constraints can either be manipulated by accessing attributed variables or by adding *constraint handling rules* to a program. The former approach of attributed variables can be much more efficient than constraint handling rules (which are themselves implemented through attributed variables) but are much more difficult to use than constraint handling rules. These variable-based approaches differ from that of *Answer Set Programming* in which a constraint problem is formulated as a set of rules, which are consistent if a stable model can be constructed for them.

XSB supports all of these approaches. Two packages based on attributed variables are presented in this chapter: CLP(R) and the `bounds` package, which provides a simple library for handling finite domains. XSB's CHR package is described in Chapter 12, and XSB's Answer Set Programming Package, `XASP` is described in Chapter 18.

Before describing the individual packages, we note that these packages can be freely used with variant tabling, the mechanisms for which handle attributed variables. However in Version 4.0, calling a predicate P that is tabled using call subsumption will raise an error if the call to P contains any constrained variables (attributed variables).

11.1 `clpr`: The CLP(R) package

The CLP(R) library supports solutions of linear equations and inequalities over the real numbers and the lazy treatment of nonlinear equations [1]. In displaying sets of equations and

[1]The CLP(R) package is based on the clpqr package included in SWI Prolog version 5.6.49. This package was originally written by Christian Holzbaur and ported to SWI by Leslie De Konick. Theresa Swift ported

disequations, the library removes redundancies, performs projections, and provides for linear optimization. The goal of the XSB port is to provide the same CLP(R) functionality as in other platforms, but also to allow constraints to be used by tabled predicates. This section provides a general introduction to the CLP(R) functionality available in XSB, for further information on the API described in Section 11.1.1 see `http://www.ai.univie.ac.at/clpqr`, or the Sicstus Prolog manual (the CLP(R) library should behave similarly on XSB and Sicstus at the level of this API).

The `clpr` package may be loaded by the command `[clpr]`. Loading the package imports exported predicates from the various files in the `clpr` package into `usermod` (see Volume 1, Section 3.3) so that they may be used in the interpreter. Modules that use the exported predicates need to explicitly import them from the files in which they are defined (e.g. `bv`, as shown below).

XSB's tabling engine supports the use of attributed variables (cf. Volume I: Library Utilities), which in turn have been used to port real constraints to XSB under the CLP(R) library of Christian Holzbauer [6]. Constraint equations are represented using the Prolog syntax for evaluable functions (Volume 1, Section 6.2.1). Formally:

ConstraintSet –>	*C* | *C* , *C*	
C –>	*Expr* `=:=` *Expr*	equation
	| *Expr* `=` *Expr*	equation
	| *Expr* < *Expr*	strict inequation
	| *Expr* > *Expr*	strict inequation
	| *Expr* `=<` *Expr*	nonstrict inequation
	| *Expr* `>=` *Expr*	nonstrict inequation
	| *Expr* `=/=` *Expr*	disequation
Expr –>	*variable*	Prolog variable
	| *number*	floating point number
	| `+` *Expr*	
	| `-` *Expr*	
	| *Expr* `+` *Expr*	
	| *Expr* `-` *Expr*	
	| *Expr* `*` *Expr*	
	| *Expr* `/` *Expr*	
	| `abs(`*Expr*`)`	
‘	| `sin(`*Expr*`)`	
	| `cos(`*Expr*`)`	
	| `tan(`*Expr*`)`	
	| `pow(`*Expr*`,`*Expr*`)`	raise to the power

the package to XSB and and wrote this XSB manual section.

```
   :- import {}/1 from clpr.

   root(N, R) :-
   root(N, 1, R).
   root(0, S, R) :- !, S=R.
   root(N, S, R) :-
N1 is N-1,
{ S1 = S/2 + 1/S },
root(N1, S1, R).
```

Figure 11.1: Example of a file with a CLP(R) predicate

`exp(`*Expr*`,`*Expr*`)`	raise to the power
`min(`*Expr*`,`*Expr*`)`	minimum of two expressions
`max(`*Expr*`,`*Expr*`)`	maximum of two expressions
`#`*(Expr)*	symbolic numerical constants

11.1.1 The CLP(R) API

From the command line, it is usually easiest to load the `clpr` package and call the predicates below directly from `usermod` (the module implicitly used by the command line). However, when calling any of these predicates from compiled code, they must be explicitly imported from their modules (e.g. `{}` must be explicitly imported from `clpr`). Figure 11.1.1 shows an example of how this is done. ‘

`{+Constraints}` `module: clpr`

When the CLP(R) package is loaded, inclusion of equations in braces (`{}`) adds `Constraints` to the constraint store where they are checked for satisfiability.

Example:

```
| ?- [clpr].
[clpr loaded]
[itf loaded]
[dump loaded]
[bv_r loaded]
[nf_r loaded]

yes

| ?- {X = Y+1, Y = 3*X}.

X = -0.5000
```

```
Y = -1.5000;

yes
```

Error Cases

- `Constraints` is not instantiated
  - `instantiation_error`
- `Constraints` is not an equation, an inequation or a disequation
  - `domain_error('constraint relation',Rel)`
- `Constraints` contains an expression `Expr` that is not a numeric expression
  - `domain_error('numeric expression',Expr)`

`entailed(+Constraint)` module: `clpr`

Succeeds if `Constraint` is logically implied by the current constraint store. `entailed/1` does not change the constraint store.

Example:

```
| ?- {A =< 4},entailed(A =\= 5).
 { A =< 4.0000 }

yes
```

Error Cases

- `Constraints` is not instantiated
  - `instantiation_error`
- `Constraints` is not an equation, an inequation or a disequation
  - `domain_error('constraint relation',Rel)`

`inf(+Expr,-Val)` clpr

`sup(+Expr,-Val)` clpr

`minimize(Expr)` clpr

`maximize(Expr)` module: `clpr`

These four related predicates provide various mechanisms to compute the maximum and minimum of expressions over variables in a constraint store. In the case where the expression is not bounded from above over the reals `sup/2` and `maximize/1` will fail; similarly if the expression is not bounded from below `inf/2` and `minimize/1` will fail.

Examples:

```
| ?- {X = 2*Y,Y >= 7},inf(X,F).
 { X >= 14.0000 }
 { Y = 0.5000 * X }

X = _h8841
Y = _h9506
F = 14.0000

| ?- {X = 2*Y,Y >= 7},minimize(X).
X = 14.0000
Y = 7.0000

| ?- {X = 2*Y,Y =< 7},maximize(X-2).

X = 14.0000
Y = 7.0000

| ?- {X = 2*Y,Y =< 7},sup(X-2,Z).
 { X =< 14.0000 }
 { Y = 0.5000 * X }

X = _h8975
Y = _h9640
Z = 12.0000

yes
| ?- {X = 2*Y,Y =< 7},maximize(X-2).

X = 14.0000
Y = 7.0000

yes
```

`inf(+Expr,-Val, +Vector, -Vertex)` clpr

`sup(+Expr,-Val, +Vector, -Vertex)` module: clpr

These predicates work like `inf/2` and `sup/2` with the following addition. `Vector` is a list of Variables, and for each variable V in `Vector`, the value of V at the extremal point `Val` is returned in corresponding position in the list `Vertex`.

Example:

```
| ?= { 2*X+Y =< 16, X+2*Y =< 11,X+3*Y =< 15, Z = 30*X+50*Y},
     sup(Z, Sup, [X,Y], Vertex).
 { X + 3.0000 * Y =< 15.0000 }
 { X + 0.5000 * Y =< 8.0000 }
```

```
 { X + 2.0000 * Y =< 11.0000 }
 { Z = 30.0000 * X + 50.0000 * Y }

X = _h816
Y = _h869
Z = _h2588
Sup = 310.0000
Vertex = [7.0000,2.0000]
```

`bb_inf(+IntegerList,+Expr,-Inf,-Vertex, +Eps)` module: clpr

Works like `inf/2` in `Expr` but assumes that all the variables in `IntegerList` have integral values. `Eps` is a positive number between 0 and 0.5 that specifies how close an element of `IntegerList` must be to an integer to be considered integral – i.e. for such an `X`, `abs(round(X) - X) < Eps`. Upon success, `Vertex` is instantiated to the integral values of all variables in `IntegerList`. `bb_inf/5` works properly for non-strict inequalities only.

Example:

```
| ?- {X > Y + Z,Y > 1, Z > 1},bb_inf([Y,Z],X,Inf,Vertex,0).
 { Z > 1.0000 }
 { Y > 1.0000 }
 { X - Y - Z > 0.0000 }

X = _h14286
Y = _h10914
Z = _h13553
Inf = 4.0000
Vertex = [2.0000,2.0000]

yes
```

Error Cases

- `IntegerList` is not instantiated
  - `instantiation_error`

`bb_inf(+IntegerList,+Expr,-Inf)` module: clpr

Works like `bb_inf/5`, but with the neighborhood, `Eps`, set to `0.001`.

Example

```
|?- {X >= Y+Z, Y > 1, Z > 1}, bb_inf([Y,Z],X,Inf)
 { Z > 1.0000 }
 { Y > 1.0000 }
```

```
 { X - Y - Z >= 0.0000 }

X = _h14289
Y = _h10913
Z = _h13556
Inf = 4.

yes
```

`dump(+Variables,+NewVars,-CodedVars` module: dump

For a list of variables `Variables` and a list of variable names `NewVars`, returns in `CodedVars` the constraints on the variables, without affecting the constraint store.

Example:

```
| ?- {X > Y+1, Y > 2},
     dump([X,Y], [x,y], CS).
 { Y > 2.0000 }
 { X - Y > 1.0000 }

X = _h17748
Y = _h17139
CS = [y > 2.0000,x - y > 1.0000];
```

Error Cases

- `Variables` is not instantiated to a list of variables
  - `instantiation_error`

`projecting_assert(+Clause)` module: dump

In XSB, when a subgoal is tabled, the tabling system automatically determines the relevant projected constraints for an answer and copies them into and out of a table. However, when a clause with constrained variables is asserted, this predicate must be used rather than `assert/1` in order to project the relevant constraints. This predicate works with either standard or trie-indexed dynamic code.

Example:

```
| ?- {X > 3},projecting_assert(q(X)).
 { X > 3.0000 }

X = _h396

yes
| ?- listing(q/1).
```

```
q(A) :-
    clpr : {A > 3.0000}.

yes
| ?- q(X),entailed(X > 2).
 { X > 3.0000 }

X = _h358

yes
| ?- q(X),entailed(X > 4).

no
```

11.2 The bounds Package

Version 4.0 of XSB does not support a full-fledged CLP(FD) package. However it does support a simplified package that maintains an upper and lower bound for logical variables. `bounds` can thus be used for simple constraint problems in the style of finite domains, as long as these problems that do not rely on too heavily on propagation of information about constraint domains [2]

Perhaps the simplest way to explain the functionality of `bounds` is by example. The query

```
|?- X in 1..2,X #> 1.
```

first indicates via `X in 1..2` that the lower bound of `X` is `1` and the higher bound `2`, and then constrains `X`, which is not yet bound, to be greater than `1`. Applying this latter constraint to `X` forces the lower bound to equal the upper bound, instantiating `X`, so that the answer to this query is `X = 2`.

Next, consider the slightly more complex query

```
|?- X in 1..3,Y in 1..3,Z in 1..3,all_different([X,Y,Z]),X = 1, Y = 2.
```

`all_different/3` constraints `X`, `Y` and `Z` each to be different, whatever their values may be. Accordingly, this constraint together with the bound restrictions, implies that instantiating `X` and `Y` also causes the instantiation of `Z`. In a similar manner, the query

```
|?- X in 1..3,Y in 1..3,Z in 1..3,sum([X,Y,Z],#=,9),
```

[2] The `bounds` package was written by Tom Schrijvers, and ported to XSB from SWI Prolog version 5.6.49 by Theresa Swift, who also wrote this manual section.

onstrains the sum of the three variables to equal `9` – and in this case assigns them a concrete value due to their domain restrictions.

In many constraint problems, it does not suffice to know whether a set of constraints is satisfiable; rather, concrete values may be needed that satisfy all constraints. One way to produce such values is through the predicate `labelling/2`

```
|?- X in 1..5,Y in 1..5,X #< Y,labeling([max(X)],[X,Y]))
```

In this query, it is specified that `X` and `Y` are both to be instantiated not just by any element of their domains, but by a value that assigns `X` to be the maximal element consistent with the constraints. Accordingly `X` is instantiated to `4` and `Y` to `5`.

Because constraints in `bounds` are based on attributed variables which are handled by XSB's variant tabling mechanisms, constrained variables can be freely used with variant tabling as the folowing fragment shows:

```
table_test(X):- X in 2..3,p(X).

:- table p/1.
p(X):- X in 1..2.

?- table_test(Y).

Y = 2
```

For a more elaborate example, we turn to the *SEND MORE MONEY* example, , in which the problem is to assign numbers to each of the letters *S,E,N,D,M,O,R,Y* so that the number *SEND* plus the number *MORE* equals the number *MONEY*. Borrowing a solution from the SWI manual [22], the `bounds` package solves this problem as:

```
send([[S,E,N,D], [M,O,R,E], [M,O,N,E,Y]]) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Carries = [C1,C2,C3,C4],
    Digits in 0..9,
    Carries in 0..1,
    M #= C4,
    O + 10 * C4 #= M + S + C3,
    N + 10 * C3 #= O + E + C2,
    E + 10 * C2 #= R + N + C1,
    Y + 10 * C1 #= E + D,
    M #>= 1,
    S #>= 1,
    all_different(Digits),
    label(Digits).
```

In many cases, it may be useful to test whether a given constraint is true or false. This can be done by unifying a variable with the truth value of a given constraint – i.e. by *reifying* the constraint. As an example, the query

```
|?-  X in 1..10, Y in 1..10,Z in 0..1,X #< Y, X #= Y #<=> Z,label([Z]).
```

sets the bounded variable `Z` to the truth value of `X #= Y`, or `0` [3].

A reader familiar with the finite domain library of Sicstus [8] will have noticed that the syntax of `bounds` is consistent with that library. It is important to note however, `bounds` maintains only the upper and lower bounds of a variables as its attributes, (along, of course with constraints on those variables) rather than an explicit vector of permissable values. As a result, `bounds` may not be suitable for large or complex constraint problems.

11.2.1 The `bounds` API

Note that `bounds` does not perform error checking, but instead relies on the error checking of lower-level comparison and arithmetic operators.

`in(-Variable,+Bound)` bounds

Adds the constraint `Bound` to `Variable`, where `Bound` should be of the form `Low..High`, with `Low` and `High` instantiated to integers. This constraint ensures that any value of `Variable` must be greater than or equal to `Low` and less than or equal to `High`. Unlike some finite-domain constraint systems, it does *not* materialize a vector of currently allowable values for `Variable`.

Variables that have not had their domains explicitly constrained are considered to be in the range `min_integer..max_integer`.

`#>(Expr1,Expr2)` bounds

`#<(Expr1,Expr2)` bounds

`#>=(Expr1,Expr2)` bounds

`#=<(Expr1,Expr2)` bounds

`#=(Expr1,Expr2)` bounds

`#=(Expr1,Expr2)` bounds

Ensures that a given relation holds between `Expr1` and `Expr2`. Within these constraints, expressions may contain the functions `+/2`, `-/2`, `*/2`, `+/2`, `+/2`, `+/2`, `mod/2`, and `abs/1` in addition to integers and variables.

`#<=>(Const1,Const2)` bounds

[3] The current version of the `bounds` package does not always seem to propagate entailment into the values of reified variables.

`#=>(Const1,Const2)` bounds

`#<=(Const1,Const2)` bounds

Constrains the truth-value of `Const1` to have the speficied logical relation ("iff", "only-if" or "if") to `Const2`, where `Const1` and `Const2` have one of the six relational operators above.

`all_different(+VarList)` bounds

`VarList` must be a list of variables: constrains all variables in `VarList` to have different values.

`sum(VarList,Op,?Value)` bounds

`VarList` must be a list of variables and `Value` an integer or variable: constrains the sum of all variables in `VarList` to have the relation `Op` to `Value` (see preceding example).

`labeling(+Opts,+VarList` bounds

This predicate succeeds if it can assign a value to each variable in `VarList` such that no constraint is violated. Note that assigning a value to each constrained variable is equivalent to deriving a solution that satisfies all constraints on the variables, which may be intractible depending on the constraints. `Opts` allows some control over how value assignment is performed in deriving the solution.

- `leftmost` Assigns values to variables in the order in which they occur. For example the query:

```
|?- X in 1..4,Y in 1..3,X #< Y,labeling([leftmost],[X,Y]),writeln([X,Y]),fail.
[1,2]
[1,3]
[2,3]

no
```

  instantiates `X` and `Y` to all values that satisfy their constraints, and does so by considering each value in the domain of `X`, checking whether it violates any constraints, then considering each value of `Y` and checking whether it violates any constraints.

- `ff` This "first-fail" strategy assignes values to variables based on the size of their domains, from smallest to largest. By adopting this strategy, it is possible to perform a smaller search for a satisfiable solution because the most constrained variables may be considered first (though the bounds of the variable are checked rather than a vector of allowable values).

- `min` and `max` This strategy labels variables in the order of their minimal lower bound or maximal upper bound.

- `min(Expr)` and `max(Expr)` This strategy labels the variables so that their assignment causes `Expr` to have a minimal or maximal value. Consider for example how these strategies would affect the labelling of the preceding query:

```
|?- X in 1..4,Y in 1..3,X #< Y,labeling([min(Y)],[X,Y]),writeln([X,Y]),fail.
[1,2]

no
|?- X in 1..4,Y in 1..3,X #< Y,labeling([max(X)],[X,Y]),writeln([X,Y]),fail.
[2,3]

no
```

`label(+VarList)` bounds

Shorthand for `labeling([leftmost],+VarList)`.

`indomain(?Var)` bounds

Unifies `Var` with an element of its domain, and upon sucessive backttrakcing, with all other elements of its domain.

`serialized(+BeginList,+Durations` bounds

`serialized/2` can be useful for scheduling problems. As input it takes a list of variables or integers representing the beginnings of temporal events, along with a list of non-negative intergers indicating the duration of each event in `BeginList`. The effect of this predicate is to constrain each of the events in `BeginList` to have a start time such that their durations do not overlap. As an example, consier the query

```
|?- X in 1..10, Y in 1..10, serialized([X,Y],[8,1]),label([X,Y]),writeln((X,Y)),fail.
```

In this query event `X` is taken to have duration of `8` units, while event `Y` is taken to have duration of `1` unit. Executing this query will instantiate `X` and `Y` to many different values, such as `(1,9)`, `(1,10)`, and `(2,10)` where `X` is less than `Y`, but also `(10,1)`, `(10,2)` and many others where `Y` is less than `X`. Refining the query as

```
X in 1..10, Y in 1..10, serialized([X,Y],[8,1]),X #< Y,label([X,Y]),writeln((X,Y)),fail.
```

removes all solutions where `Y` is less than `X`.

`lex_chain(+List)` bounds

`lex_chain/1` takes as input a list of lists of variables and integers, and enforces the constraint that each element in a given list is less than or equal to the elements in all succeeding lists. As an example, consider the query

```
|?- X in 1..3,Y in 1..3,lex_chain([[X],[2],[Y]]),label([X,Y]),writeln([X,Y]),fail.
[1,2]
[1,3]
[2,2]
[2,3]
```

`lex_chain/1` ensures that `X` is less than or equal to `2` which is less than or equal to `Y`.

Chapter 12

Constraint Handling Rules

12.1 Introduction

Constraint Handling Rules (CHR) is a committed-choice bottom-up language embedded in XSB. It is designed for writing constraint solvers and is particularly useful for providing application-specific constraints. It has been used in many kinds of applications, like scheduling, model checking, abduction, type checking among many others.

CHR has previously been implemented in other Prolog systems (SICStus, Eclipse, Yap, hProlog), Haskell and Java. The XSB CHR system is based on the hProlog CHR system.

In this documentation we restrict ourselves to giving a short overview of CHR in general and mainly focus on XSB-specific elements. For a more thorough review of CHR we refer the reader to [5]. More background on CHR can be found at [4].

In Section 12.2 we present the syntax of CHR in XSB and explain informally its operational semantics. Next, Section 12.3 deals with practical issues of writing and compiling XSB programs containing CHR. Section 12.4 provides a few useful predicates to inspect the constraint store and Section 12.5 illustrates CHR with two example programs. How to combine CHR with tabled predicates is covered in Section 12.6. Finally, Section 12.7 concludes with a few practical guidelines for using CHR.

12.2 Syntax and Semantics

12.2.1 Syntax

The syntax of CHR rules in XSB is the following:

```
rules --> rule, rules.
```

```
rules --> [].

rule --> name, actual_rule, pragma, [atom('.')].

name --> xsb_atom, [atom('@')].
name --> [].

actual_rule --> simplification_rule.
actual_rule --> propagation_rule.
actual_rule --> simpagation_rule.

simplification_rule --> constraints, [atom('<=>')], guard, body.
propagation_rule --> constraints, [atom('==>')], guard, body.
simpagation_rule --> constraints, [atom('\')], constraints, [atom('<=>')],
                     guard, body.

constraints --> constraint, constraint_id.
constraints --> constraint, [atom(',')], constraints.

constraint --> xsb_compound_term.

constraint_id --> [].
constraint_id --> [atom('#')], xsb_variable.

guard --> [].
guard --> xsb_goal, [atom('|')].

body --> xsb_goal.

pragma --> [].
pragma --> [atom('pragma')], actual_pragmas.

actual_pragmas --> actual_pragma.
actual_pragmas --> actual_pragma, [atom(',')], actual_pragmas.

actual_pragma --> [atom('passive(')], xsb_variable, [atom(')')].
```

Additional syntax-related terminology:

- **head:** the constraints in an `actual_rule` before the arrow (either `<=>` or `==>`)

12.2.2 Semantics

In this subsection the operational semantics of CHR in XSB are presented informally. They do not differ essentially from other CHR systems.

When a constraint is called, it is considered an active constraint and the system will try to apply the rules to it. Rules are tried and executed sequentially in the order they are written.

A rule is conceptually tried for an active constraint in the following way. The active constraint is matched with a constraint in the head of the rule. If more constraints appear in the head they are looked for among the suspended constraints, which are called passive constraints in this context. If the necessary passive constraints can be found and all match with the head of the rule and the guard of the rule succeeds, then the rule is committed and the body of the rule executed. If not all the necessary passive constraint can be found, the matching fails or the guard fails, then the body is not executed and the process of trying and executing simply continues with the following rules. If for a rule, there are multiple constraints in the head, the active constraint will try the rule sequentially multiple times, each time trying to match with another constraint.

This process ends either when the active constraint disappears, i.e. it is removed by some rule, or after the last rule has been processed. In the latter case the active constraint becomes suspended.

A suspended constraint is eligible as a passive constraint for an active constraint. The other way it may interact again with the rules, is when a variable appearing in the constraint becomes bound to either a non-variable or another variable involved in one or more constraints. In that case the constraint is triggered, i.e. it becomes an active constraint and all the rules are tried.

Rule Types There are three different kinds of rules, each with their specific semantics:

- `simplification:`

  The simplification rule removes the constraints in its head and calls its body.

- `propagation:`

  The propagation rule calls its body exactly once for the constraints in its head.

- `simpagation:`

  The simpagation rule removes the constraints in its head after the \ and then calls its body. It is an optimization of simplification rules of the form:

$$constraints_1, constraints_2 <=> constraints_1, body$$

Namely, in the simpagation form:

$$constraints_1 \backslash constraints_2 <=> body$$

The $constraints_1$ constraints are not called in the body.

Rule Names Naming a rule is optional and has no semantical meaning. It only functions as documentation for the programmer.

Pragmas The semantics of the pragmas are:

- **passive/1:** the constraint in the head of a rule with the identifier specified by the **passive/1** pragma can only act as a passive constraint in that rule.

Additional pragmas may be released in the future.

12.3 CHR in XSB Programs

12.3.1 Embedding in XSB Programs

Since chr is an XSB package, it must be explicitly loaded before being used.

```
?- [chr].
```

CHR rules are written in a tt .chr file. They should be preceded by a declaration of the constraints used:

```
:- constraints ConstraintSpec1, ConstraintSpec2, ...
```

where each `ConstraintSpec` is a functor description of the form name/arity pair. Ordinary code may be freely written between the CHR rules.

The CHR constraints defined in a particular .chr file are associated with a CHR module. The CHR module name can be any atom. The default module is `user`. A different module name can be declared as follows:

```
:- chr_module(modulename).
```

One should never load different files with the same CHR module name.

12.3.2 Compilation

Files containing CHR rules are required to have a `.chr` extension, and their compilation has two steps. First the `.chr` file is preprocessed into a `.P` file containing XSB code. This `.P` file can then be loaded in the XSB emulator and used normally.

`load_chr(File)` chr_pp

`load_chr/1` takes as input a file name whose extension is either `.chr` or that has no extension. It preprocesses `File` if the times of the CHR rule file is newer than that of the corresponding Prolog file, and then consults the Prolog file.

`preprocess(File,PFile)` chr_pp

`preprocess/2` takes as input a file name whose extension is either `.chr` or that has no extension. It preprocesses `File` if the times of the CHR rule file is newer than that of the corresponding Prolog file, but does not consult the Prolog file.

12.4 Useful Predicates

The `chr` module contains several useful predicates that allow inspecting and printing the content of the constraint store.

`show_store(+Mod)` chr

Prints all suspended constraints of module `Mod` to the standard output.

`suspended_chr_constraints(+Mod,-List)` chr

Returns the list of all suspended CHR constraints of the given module.

12.5 Examples

Here are two example constraint solvers written in CHR.

- The program below defines a solver with one constraint, `leq/2`, which is a less-than-or-equal constraint.

```
:- chr_module(leq).

:- export cycle/3.

:- import length/2 from basics.
```

```
:- constraints leq/2.
reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).

cycle(X,Y,Z):-
        leq(X,Y),
        leq(Y,Z),
        leq(Z,X).
```

- The program below implements a simple finite domain constraint solver.

```
:- chr_module(dom).

:- import member/2 from basics.

:- constraints dom/2.

dom(X,[]) <=> fail.
dom(X,[Y]) <=> X = Y.
dom(X,L1), dom(X,L2) <=> intersection(L1,L2,L3), dom(X,L3).

intersection([],_,[]).
intersection([H|T],L2,[H|L3]) :-
        member(H,L2), !,
        intersection(T,L2,L3).
intersection([_|T],L2,L3) :-
        intersection(T,L2,L3).
```

These and more examples can be found in the `examples/chr/` folder accompanying this XSB release.

12.6 CHR and Tabling

The advantage of CHR in XSB over other Prolog systems, is that CHR can be combined with tabling. Hence part of the constraint solving can be performed once and reused many times. This has already shown to be useful for applications of model checking with constraints.

However the use of CHR constraints is slightly more complicated for tabled predicates. This section covers how exactly to write a tabled predicate that has one or more arguments

that also appear as arguments in suspended constraints. In the current release the CHR-related parts of the tabled predicates have to be written by hand. In a future release this may be substituted by an automatic transformation.

12.6.1 General Issues and Principles

The general issue is how call constraints should be passed in to the tabled predicate and how answer constraints are passed out of the predicate. Additionally, in some cases care has to be taken not to generate infinite programs.

The recommended approach is to write the desired tabled predicate as if no additional code is required to integrate it with CHR. Next transform the tabled predicate to take into account the combination of tabling and CHR. Currently this transformation step has to be done by hand. In the future we hope to replace this hand coding with programmer declarations that guide automated transformations.

Hence we depart from an ordinary tabled predicate, say `p/1`:

```
:- table p/1.

p(X) :-
   ... /* original body of p/1 */.
```

In the following we will present several transformations or extensions of this code to achieve a particular behavior. At least the transformation discussed in subsection 12.6.2 should be applied to obtain a working integration of CHR and tabling. Further extensions are optional.

12.6.2 Call Abstraction

Currently only one type of call abstraction is supported: full constraint abstraction, i.e. all constraints on variables in the call should be removed. The technique to accomplish this is to replace all variables in the call that have constraints on them with fresh variables. After the call, the original variables should be unified with the new ones.

In addition, the call environment constraint store should be replaced with an empty constraint store before the call and on return the answer store should be merged back into the call environment constraint store.

The previously mentioned tabled predicate `p/1` should be transformed to:

```
:- import merge_answer_store/1,
          get_chr_store/1,
```

```
          set_chr_store/1,
          get_chr_answer_store/2
   from chr.

:- table tabled_p/2.

p(X) :-
        tabled_p(X1,AnswerStore),
        merge_answer_store(AnswerStore),
        X1 = X.

tabled_p(X,AnswerStore) :-
        get_chr_store(CallStore),
        set_chr_store(_EmptyStore)
        orig_p(X),
        get_chr_answer_store(chrmod,AnswerStore),
        set_chr_store(CallStore).

orig_p(X) :-
   ... /* original body of p/1 */.
```

This example shows how to table the CHR constraints of a single CHR module `chrmod`. If multiple CHR modules are involved, one should add similar arguments for the other modules.

12.6.3 Answer Projection

To get rid of irrelevant constraints, most notably on local variables, the answer constraint store should in some cases be projected on the variables in the call. This is particularly important for programs where otherwise an infinite number of answers with ever growing answer constraint stores could be generated.

The current technique of projection is to provide an additional `project/1` constraint to the CHR solver definition. The argument of this constraint is the list of variables to project on. Appropriate CHR rules should be written to describe the interaction of this `project/1` constraint with other constraints in the store. An additional rule should take care of removing the `project/1` constraint after all such interaction.

The `project/1` constraint should be posed before returning from the tabled predicate.

If this approach is not satisfactory or powerful enough to implement the desired projection operation, you should resort to manipulating the underlying constraint store representation. Contact the maintainer of XSB's CHR system for assistance.

Example Take for example a predicate `p/1` with a less than or equal constraint `leq/2` on variables and integers. The predicate `p/1` has local variables, but when `p` returns we are not interested in any constraints involving local variables. Hence we project on the argument of `p/1` with a project constraint as follows:

```
:- import memberchk/2 from lists.

:- import merge_answer_store/1,
          get_chr_store/1,
          set_chr_store/1,
          get_chr_answer_store/2
   from chr.

:- table tabled_p/2.

:- constraints leq/2, project/1.

... /* other CHR rules */
project(L) \ leq(X,Y) <=>
        ( var(X), \+ memberchk(X,L)
        ; var(Y), \+ memberchk(Y,L)
        ) | true.

project(_) <=> true.

p(X) :-
        tabled_p(X1,AnswerStore),
        merge_answer_store(AnswerStore),
        X1 = X.

tabled_p(X,AnswerStore) :-
        get_chr_store(CallStore),
        set_chr_store(_EmptyStore)
        orig_p(X),
        project([X]),
        get_chr_answer_store(chrmod,AnswerStore),
        set_chr_store(CallStore).

orig_p(X) :-
   ... /* original body of p/1 */.
```

The example in the following subsection shows projection in a full application.

12.6.4 Answer Combination

Sometimes it is desirable to combine different answers to a tabled predicate into one single answer or a subset of answers. Especially when otherwise there would be an infinite number of answers. If the answers are expressed as constraints on some arguments and the logic of combining is encoded as CHR rules, answers can be combined by merging the respective answer constraint stores.

Another case where this is useful is when optimization is desired. If the answer to a predicate represents a valid solution, but an optimal solution is desired, the answer should be represented as constraints on arguments. By combining the answer constraints, only the most constrained, or optimal, answer is kept.

Example An example of a program that combines answers for both termination and optimisation is the shortest path program below:

```
:- chr_module(path).

:- import length/2 from lists.

:- import merge_chr_answer_store/1,
          get_chr_store/1,
          set_chr_store/1,
          get_chr_answer_store/2
   from chr.

breg_retskel(A,B,C,D) :- '_$builtin'(154).

:- constraints geq/2, plus/3, project/1.

geq(X,N) \ geq(X,M) <=> number(N), number(M), N =< M | true.

reflexivity  @ geq(X,X) <=> true.
antisymmetry @ geq(X,Y), geq(Y,X) <=> X = Y.
idempotence  @ geq(X,Y) \ geq(X,Y) <=> true.
transitivity @ geq(X,Y), geq(Y,Z) ==> var(Y) | geq(X,Z).

plus(A,B,C) <=> number(A), number(B) | C is A + B.
plus(A,B,C), geq(A,A1) ==> plus(A1,B,C1), geq(C,C1).
plus(A,B,C), geq(B,B1) ==> plus(A,B1,C1), geq(C,C1).

project(X) \ plus(_,_,_) # ID <=> true pragma passive(ID).
project(X) \ geq(Y,Z) # ID <=> (Y \== X ; var(Z) )| true pragma passive(ID).
project(_) <=> true.
```

```
path(X,Y,C) :-
tabled_path(X,Y,C1,AS),
merge_chr_answer_store(AS),
C = C1.

:- table tabled_path/4.

tabled_path(X,Y,C,AS) :-
'_$savecp'(Breg),
breg_retskel(Breg,4,Skel,Cs),
copy_term(p(X,Y,C,AS,Skel),p(OldX,OldY,OldC,OldAS,OldSkel)),
        get_chr_store(GS),
set_chr_store(_GS1),
orig_path(X,Y,C),
        project(C),
( get_returns(Cs,OldSkel,Leaf),
  OldX == X, OldY == Y ->
            merge_chr_answer_store(OldAS),
            C = OldC,
            get_chr_answer_store(path,MergedAS),
            sort(MergedAS,AS),
            ( AS = OldAs ->
                fail
            ;
                delete_return(Cs,Leaf)
            )
;
            get_chr_answer_store(path,UnsortedAS),
            sort(UnsortedAS,AS)
),
        set_chr_store(GS).

orig_path(X,Y,C) :- edge(X,Y,C1), geq(C,C1).
orig_path(X,Y,C) :- path(X,Z,C2), edge(Z,Y,C1), plus(C1,C2,C0), geq(C,C0).

edge(a,b,1).
edge(b,a,1).
edge(b,c,1).
edge(a,c,3).
edge(c,a,1).
```

The predicate `orig_path/3` specifies a possible path between two nodes in a graph. In `tabled_path/4` multiple possible paths are combined together into a single path with the shortest distance. Hence the tabling of the predicate will reject new answers that have a

worse distance and will replace the old answer when a better answer is found. The final answer gives the optimal solution, the shortest path. It is also necessary for termination to keep only the best answer. When cycles appear in the graph, paths with longer and longer distance could otherwise be put in the table, contributing to the generation of even longer paths. Failing for worse answers avoids this infinite build-up.

The predicate also includes a projection to remove constraints on local variables and only retain the bounds on the distance.

The sorting canonicalizes the answer stores, so that they can be compared.

12.6.5 Overview of Tabling-related Predicates

`merge_answer_store(+AnswerStore)` chr
: Merges the given CHR answer store into the current global CHR constraint store.

`get_chr_store(-ConstraintStore)` chr
: Returns the current global CHR constraint store.

`set_chr_store(?ConstraintStore)` chr
: Set the current global CHR constraint store. If the argument is a fresh variable, the current global CHR constaint store is set to be an empty store.

`get_chr_answer_store(+Mod,-AnswerStore)` chr
: Returns the part of the current global CHR constraint store of constraints in the specified CHR module, in the format of an answer store usable as a return argument of a tabled predicate.

12.7 Guidelines

In this section we cover several guidelines on how to use CHR to write constraint solvers and how to do so efficiently.

- **Set semantics:** The CHR system allows the presence of identical constraints, i.e. multiple constraints with the same functor, arity and arguments. For most constraint solvers, this is not desirable: it affects efficiency and possibly termination. Hence appropriate simpagation rules should be added of the form:

$$constraint \backslash constraint <=> true$$

- **Multi-headed rules:** Multi-headed rules are executed more efficiently when the constraints share one or more variables.

12.8 CHRd

An alternate implementation of CHR can be found in the CHRd package. The main objective of the CHRd package is to optimize processing of constraints in the environment where termination is guaranteed by the tabling engine, (and where termination benefits provided by the existing solver are not critical). CHRd takes advantage of XSB's tabling to simplify CHR's underlying storage structures and solvers. Specifically, we entirely eliminate the thread-global constraint store in favor of a distributed one, realized as a collection of sets of constraints entirely associated with program variables. This decision limits the applicability of CHRd to a restricted class of CHR programs, refered to as direct-indexed CHR,in which all constraints in the head of a rule are connected by shared variables. Most CHR programs are direct-indexed, and other programs may be easily converted to fall into this class. Another advance of CHRd is its set-based semantics which removes the need to maintain the propagation history, thus allowing further simplicity in the representation of the constraints. The CHRd package itself is described in [13], and both the semantics of CHRd and the class of direct-indexed CHR are formally defined in [14].

Chapter 13

The `viewsys` Package

chapter:viewsys

By David S Warren

The `viewsys` package provides a powerful mechanism to support tasks information is combined from different sources. Views can be constructed either from external data or from other views. In this way, a *View System* supports a DAG of views.

More precisely we can think of a view as an abstracted data source – say a web query or database query. Base views are data sources from outside the system. A non-base view is a data source that is determined (and computed) by its process applied to its input data sources. An example of a non-base view might consist of data from two sources where information from one source may override that of another source under certain conditions.

A view system workflow (*ViewSys* for short) describes the names of the views, their input views, the command to be run to generate a view from its inputs, etc. A particular *instance* of a ViewSys is determined by the specific external data sources associated with the base views of the ViewSys. An *instance* of a Viewsys designates the set of views constructed from a given set of (external) base views at a given time. It is useful to give names to such instances, usually indicating the external source of the base data sources.

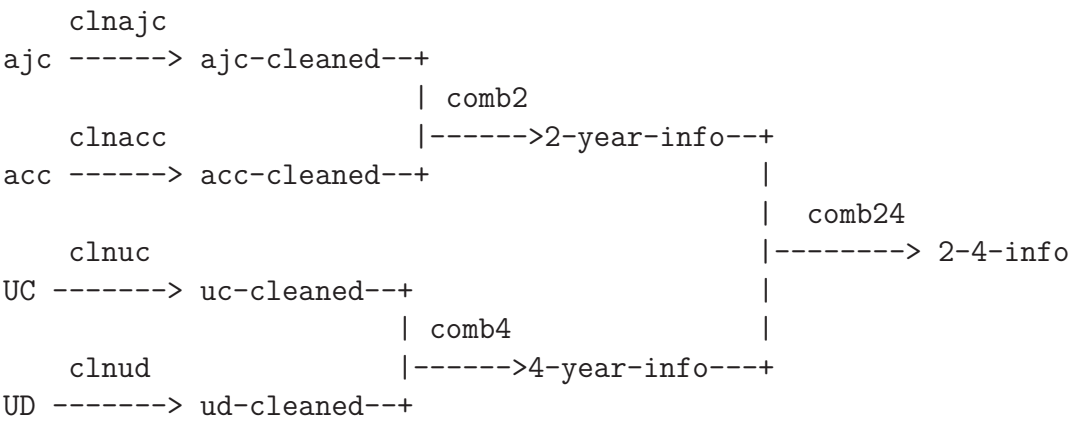
Another useful component of a view system is what is called a *consistency view*. The purpose of a consistency view is to check to see whether a regular view is 'consistent'. The command for a consistency view should return non-zero if the view instance is not deemed to be consistent. The view system will run consistency views where applicable and will not use a view as input to another view that it supports if it is deemed not consistemt. A single view may have zero or more consistency views associated with it.

13.1 An Example

Consider a situation in which we are collecting data from four institutions of higher education and want to integrate that data into a dataset that allows us to make coherent queries across the data from all institutions. We might want to answer questions about the possibility of transferring classes between the schools, or perhaps whether a student might take a class scheduled at one that would be equivalent to one at another, if schedules don't conflict.

Say we have two community colleges, AJC and ACC, and two 4-year colleges, UC and UD. And we collect information from each of them concerning, say, currently scheduled classes at their institutions.

To integrate data from all four institutions, we might create the following view system:



For each raw-data input from an institution, we have a process to "clean" that data (indicated in the diagram by a name `cln<inst>`), that generates a file (view) containing "cleaned"and "standardized" data (indicated in the diagram by `<inst>-cleaned`.) Then we have a process, `comb2`, that combines the two cleaned community college datasets to create a view, 2-year-info; and another, `comb4`, that combines the two cleaned 4-year college datasets to create the view, 4-year-info. And finally, we have a process, `comb24`, that takes those two views and generates a fully combined dataset (i.e., view), 2-4-info.

This viewsys system has 11 views, 4 of which are base views and 7 derived views. And it has 7 processes, one to generate each derived view.

We can imagine what these processes might do: the cleaning processes would do institution-specific transformations of the input data, maybe standardizing names of equivalent classes; inferring a new variable of the level of the classe (intro, intermediate, advanced) from the class naming/numbering conventions of the particular institution; standardizing class-time representations given different scheduling conventions; etc. The `comb<?>` processes might simply project and union their inputs, but in the real world, they are more likely to perform other more complex inferences and transformations.

We could easily imagine having other (mostly static) data inputs (not shown here) to these cleaning processes that provide institution-specific information necessary to do such transformations. We can also imagine that we have another process that uses, say, the 2-year-info view, to combine it with other information we've gleaned from 2-year colleges to provide another view that can answer other questions of interest.

We can imagine that the datasets we get from the source institutions arrive at different times but we want the best data in the coherent eviews to be available to any query. So if a new file from, say, `UC`, shows up, we need only run the processes `clnuc`, `comb4`, and `comb24` to be sure that all data is up to date.

13.2 The ViewSys Data Model

A ViewSys workflow is specified by a set of facts of the following predicates. Users should put the appropriate facts for these predicates that define their view system into a file named `viewsys_view_info.P`.

View Framework Model

For each view (base or derived), there is a `view/6` fact that describes it:

`view(View,Type,ViewNameTemplate,[InputViews],[Opts],ShCmd)` where:

- `View` is the name of the view;
- `Type` is `file`, `dir(<FileNames>)`, or `table`. <If it is `file`, the view is stored in a file (that is generated by the `ShCmd`). It `dir(<FileNames>)` the view is stored in multiple files in a directory. `<FileNames>` are the (relative) names of the files that store the view in that directory (instance). Finally, if the type is `table`, the view is a database table.
- `ViewNameTemplate` is the path template for where instance versions are stored. This template string normally contains the pattern variable $INSTANCE$ which will be replaced by the instance name to obtain the name of an instance of this view. (If the viewsys will have only one instance, the $INSTANCE$ variable is not required.)

  A template may also contain user-defined pattern variables of the form $USERVARNAME$ where $USERVARNAME$ is any upper-case letter sequence (except those reserved for viewsys system variables.) User-defined pattern variable values are defined in facts of the form

  `viewsys_uservar($USERVARNAME$,VarValueString)`.

  When instantiated by an instance name and user-variable values, the template identifies the instance of the given view (e.g., a file, table or directory).

- `[InputViews]` is a list of the names of views that this view directly depends on, i.e., the inputs needed to generate this view. This is an empty list for base views. Normally these input view indicators are atoms for which there is another `view/6` fact that describes it. However, if that view generates a directory and the input to this view is a file in that directory, then that filename should be put as an argument to the view atom. E.g., if the view, m_view, generates a directory and several files in it and this view needs to use the file 'first_file.P' from that directory, then the input view indicator in this list should be the term `m_view('first_file.P')`.

- `[Opts]` is a list of options. The possible options are:

  - `split(N)` where `N` is a positive integer. This tells `viewsys` to split the first input view file into `N` subfiles; to run this command on each of those subfiles; and to concatenate all the resulting subfiles back together to get the output file for this view. Of course, this is only appropriate for view commands for which this process gives the same answer as running it on the large unsplit file. When the command satisfies this property, this option can allow the records in a large file to be processed in parallel.

    If this option is used, the user must first run `expand_views(ViewDir)` to generate a viewsys file that implements the splitting. It will move the `viewsys_view_info.P` file to `viewsys_view_orig_info.P` replace it with a modified version of the file that will drive the `viewsys` processing. (If the file `viewsys_view_orig_info.P` exists, the operation will indicate an error, in order to protect against inadvertantly overwriting the original `viewsys_view_info.P` file.)

- `ShCmd` is the shell command to execute to generate the view instance from its input view instances. (Ignored for base views.) The shell command can be in one of two forms:

  1. a string containing metavariables of the form $INP1$, $INP2$, ..., and OUT, which will be replaced by the filenames of the input view instance files/directories and the output view instance file/directory, respectively; or
  2. a string containing the metavariables $INPUTFILES$ and $OUTPUTFILE$, which will be replaces with the sequence of input filenames and the output filename, respectively, where each filename is enclosed in double-quotes. This is often appropreate for shell commands. If the shell string doesn't contain any of the metavariables, then it is treated as if it were: <ShCmd> $INPUTFILES$ $OUTPUTFILE$'.

User-defined syntactic variables can be used in filename templates and in shell command templates to make it easier to define filenames and commands. The predicate `viewsys_uservar/2` is used to define user variables, and facts for this predicate should be placed in the `viewsys_view_info.P` file. For example, assume the user adds the following facts to that file:

```
viewsys_uservar('$DATA_DIR$','C:/userfiles/project1/data').
viewsys_uservar('$SCRIPT_LIB$','c:/userfiles/project1/scripts').
```

With these declarations in `viewsys_view_info.P`, a file template string could be of the form `$DATA_DIR$/data_file_13`, which after replacement of the syntactic variable by its value would refer to the file 'C:/userfiles/project1/data/data_file_13'. A shell command string could be `sh $SCRIPT_LIB$/script_cc.sh`, which after replacements would cause the command `sh c:/userfiles/project1/scripts/script_cc.sh` to be run. User variables are normally defined at the beginning of the view file and can be used to allow locations to be easily changed. The value of a user variable may contain another user variable, but, of course, cycles are not permitted.

The user must define a uservar of $STDOUTFILE$ which is the filename into which the stdout streams from the execution of a view generation will be put. The user should use the $INSTANCE$ and $VIEW$ variables to make it unique for each output stream.

Consistency Views For each consistency view, there is a `consView/5` fact:

`consView(ConsViewName, CheckedViewName, FileTemplate, [Inputs], ShCmd)`

where

- `ConsViewName` is the name of the consistency view.
- `ViewName` is the name of the view this view checks.
- `FileTemplate` is the template for the output file for this consistency check. This file may be used to provide information as to why the consistency check failed (or passed.)
- `[InputViews]` is a list of parameter input views (maybe empty)
- `ShCmd` is the shell command the executes the consistency check. The inputs are the the filename containing the view instance to be checked followd by the input view file instances. The output is the output file instance. These parameters are processed similarly to the processing for shell-commands for regular views.

13.3 View Instance Model

A ViewSys Instance is a particular instantiation of a ViewSys workflow that is identified by a name, usually indicating the source of the base views. Of course, the files (directories) that contain instances of views must all be distinct.

View instances are described by another set of facts, which are stored in a file named `viewsys_instance_info.P`. Whereas the user is responsible for creating the `viewsys_view_info.P`

file, viewsys creates and maintains the `viewsys_instance_info.P` file in response to viewsys commands entered by the user.

For each view instance (base or derived), there is a `viewInst/5` fact:

viewInst(View,InstName,Status,Date,Began) where:

- `View` is the name of a view;
- `InstName` is the name of the instance;
- `Status` is the status of this view instance `not_generated`, `being_generated(ProcName)`, `generated`, `generation_failed`. (For base view instances this is always `generated`.)
- `Date` is the date-time the view instance was generated.
- `Began` is the date-time at which the generation of this view began. (This is the same as Date above for base view instances.) It is used to estimate how long it will take to generate this view output given its inputs.

For each consistency view instance, there is a consViewInst/5 fact:

consViewInst(ConsViewName, InstName, Status, Date, Began)

where;

- `ConsViewName` is the name of the consistency view.
- `Status` is this consistency view, same as for viewInst status.
- `Date` is the date-time the check was generated.
- `Began` is the date-time at which the generation of this view began.

The ViewSys relations, `view/6`, `consView/5`, and `viewOrig/6`, are stored in the file named `viewsys_view_info.P`. It is read for most commands, but not updated. (Only `expand_views/1` generates this file from the file namsd `viewsys_view_orig_info.P`.) `viewInst/5`, and `consViewInst/5` are stored in the file named `vieewsys_instance_info.P`, and the directory containing these files is explicitly provided to predicates that need to operate on it. The contents of the files are Prolog terms in canonical form.

A lockfile (named `lock_view` in the viewsys directory) is obtained whenever these files are read, and it is kept until reading and rewriting (if necessary) is completed.

13.4 Using ViewSys

The viewsys system is normally used as follows. The user creates a directory to hold the viewsys information. She creates a file `viewsys_view_info.P` in this directory containing the desired `view/6`, and `consView/5` facts that describe the desired view system. Then the user consults the viewsys.P package, and runs `check_viewsys/1` to report any obvious inconsistencies in the view system specified in the file `viewsys_view_info.P`. After the check passes, if any views have the `split(N)` option, the user should copy the `viewsys_view_info.P` file to a file named `viewsys_orig_view_info.P` and then run `expand_views/1` to generate the appropriate file `viewsys_view_info.P` to contain the views necessary to split, execute and combine the results. This will overwrite the `viewsys_view_info.P` file. (From then on, should the viewsys need to be modified, the user should edit the `viewsys_orig_view_info.P` file, and rerun `expand_views/1` to regenerate the `viewsys_view_info.P` file.) The user will then run `generate_view_instance/2` to generate an instance (or instances) of the view system into the file `viewsys_instance_info.P`. After that the user will run `update_views/4` to run the workflow to generate all the view contents. Then the user checks the generated logging to determine if there were any errors. If so, the user corrects the programs (the viewsys specification, whatever), executes `reset_failed/2` and reruns `update_views/4`. The user can also use `viewsys_status/1` to determine what the state of the view system is, and to determine what needs to be fixed and what needs to be rerun. If the execution of `update_views/4` is aborted or somehow does not complete, the user can run `reset_unfinished/2` to reset the views that were in process, so that a subsequent `update_views/4` will try to recompute those unfinished computations.

`generate_new_instance(ViewSys,VInst)` module: `view_sys`

`generate_new_instance(+ViewSys,+VInst)` creates a brand new instance of the view system `ViewSys` named `VInst`. It generates `new viewInst/5` facts for every view (base and derived) according to the file templates defined in the `baseView/4`, and `view/6` facts of the ViewSys. `VInst` may be a list of instance names, in which case initial instances are created for each one.

`update_instance(ViewSys,VInst)` module: `view_sys`

`update_instance(+ViewSys,+VInst)` updates an instance of the view system `ViewSys` named VInst. It is similar to `generate_new_instance/2` but doesn't change existing instance records. It generates a new `viewInst/5` (or `consViewInst/5`) fact for every view (base and derived) that does not already exist in the `viewsys_instance_info.P` file. It doesn't change instances that already exisit, thus preserving their statuses and process times.

`delete_instance(ViewSys,VInst)` module: `view_sys`

`delete_instance(+ViewSys,+VInst)` removes an entire instance from the view system. Any files of view contents that have been generated remain; only information

concerning this instance in the viewsys_instance_info.P file is removed, so these view instances are no longer maintained.

update_views(ViewSys,ViewInstList,ProcName,NProcs) module: view_sys
update_views(+ViewSys, +ViewInstList, +ProcName, +NProcs) is the predicate that runs the shell commands of view instances to create view instance contents. It ensures that most recent versions of the view instances in ViewInstList (and all instances required for those views, recursively) are up to date by executing the commands as necessary. A view instance is represented in this list by a term View:InstName. If ViewInstList is the atom 'all', all view instances will be processed. This predicate will determine what computations can be done concurrently and will use up to NProcs concurrent processes (using spawn_process on the current machine) to compute them. ProcName is a user-provided process namde that used to identify this (perhaps very long-running) process; it is used to indicate, in Ststus=being_updated(ProcName) that a view instance is in the process of being computing by this update_views invocation. reset_unfinished/2 uses the name to identify the view instances that a particular invocation of this process is responsible for.

start_available_procs(ViewSys,ViewInstList,ExecutingPids,ProcName,NProcs,Slp,OStr)
module: view_sys
start_available_procs(+ViewSys, +ViewInstList, +ExecutingPids, +ProcName, +NProcs, +Slp, +OStr) is an internal predicate that supports the view_update/4 processing. It finds all views that can be generated (or checked), starts processes to compute NProcs of them, and then calls monitor_running_procs/7 to monitor their progress and start more processes as these terminate. This is an internal predicate, not available for call from outside the module. The parameters to start_available_procs/7 are:

1. ViewSys is the directory containing the viewsys_info.P file describing the view system.
2. ViewInstList is a) an explicit list of records of the form View:Inst identifying the (derived) views, normally 'root' views, that are intended to be generated by the currently running update_view/4 invocation; or b) the constant 'all' indicating that all view instances of the view system are intended to be generated.
3. ExecutingPids are pid records of the currently running processes that have been spawned. A pid record is of the form: pid(Pid,ShCmd,SStr,FileOut,Datime, View,File,Inst), where
   - Pid is the process ID of the process (as returned by spawn_process/5.)
   - ShCmd is the shell command that was used to start the process.
   - SStr is the output stream of the process's stdout and stderr file.
   - FileOut is the name of the file connected to the stdout/stderr stream.

- Datime is the datime that the process was started.
- View is the view the process is generating.
- @var(File) is the name of the output file to contain the contents of the view instance.
- Inst is the instance of the view the process is generating.

4. ProcName is the user-provided name of this entire update process, and is used to mark views (in the viewsys_instance_info.P file) during processing so they can be identified as associated to this view-update process if some error occurs.
5. NProcs is the number of 'processors' available for a process to be scheduled on. The 'processors' are virtual, and this is used to control the maximum number of concurrently running processes.
6. Slp is the number of seconds to sleep if no subprocess is available for starting before checking again to see if some subprocess has completed in the interim.
7. OStr is the output stream used to write progress messages when processes start and complete.

monitor_running_procs(Pids,NProcs,ViewSys,VInstList,ProcName,Slp,OStr) module: view_sys
monitor_running_procs(+Pids, +NProcs, +ViewSys, +VInstList, +ProcName, +Slp, +OStr) is an internal predicate that monitors previously spawned running processes, calling start_available_procs/7 to spawn new ones when running processes finish.

1. Pids is the list of process IDs of running processes. Each entry is a record of the form pid(Pid,Cmd,StdStr,FileOut,Datime,View,File,Inst) where:
   - Pid is the process ID of the process (as returned by spawn_process/5.)
   - ShCmd is the shell command that was used to start the process.
   - SStr is the output stream of the process's stdout and stderr file.
   - FileOut is the name of the file connected to the stdout/stderr stream.
   - Datime is the datime that the process was started.
   - View is the view the process is generating.
   - @var(File) is the name of the output file to contain the contents of the view instance.
   - Inst is the instance of the view the process is generating.
2. NProcs is the number of 'processors' that are currently available for use. starrt_available_procs can start up to this number of new processes.
3. ViewSys is the viewsys directory;
4. VInstList is the list of view instances (or 'all') that are being updated by this execution of update_views/4.;

5. ProcName is the caller-provided name of this update processor used to mark views that are being updated by this update process; and
6. Slp is the number of seconds to sleep if no process is available for starting.
7. OStr is the output stream for writing status messages;

generate_file_from_template(+FileTempl,+View,+Inst,-FileName) module: view_sys
generate_file_from_template(+FileTempl,+View,+Inst,-FileName) takes a file template string (with embedded $$ variable names), a view name, View, an instance name, Inst, and replaces the variable names with their values, returnning FileName.

invalidate_all_instances(ViewSys) module: view_sys
invalidate_all_instances(+ViewSys) invalidates all views, so a subsequent invocation of update_views/4 would recompute them all.).

invalidate_view_instances(ViewSys,ViewInstList) module: view_sys
invalidate_view_instances(+ViewSys,+ViewInstList) invalidates a set of view instances indicated by ViewInstList. If ViewInstList is the atom 'all', this invalidates all instances (exactly as invalidate_all_instances/1) does.) If ViewInstList is a list of terms of the form View:VInst then these indicated view instances (and all views that depend on them) will be invalidated. If ViewInstList is the atom 'filetime', then the times of the instance files will be used to invalidate view instances where the filetime of some view instance input file is later than the filetime of the view instance output file. Note this does not account for the time it takes to run the shell command that generates the view output, so for it to work, no view instance input file should be changed while a view instance is in the process of being generated.

This predicate can be used if a base instance file is replaced with a new instance. It can be used if the contents of a view instance are found not to be correct, and the generating process has been modified to fix it.

reset_unfinished(ViewSys,ProcName) module: view_sys
reset_unfinished(+ViewSys,+ProcName) resets view instances that are unfinished due to some abort, i.e., that are marked as being_generated(ProcName) after the view_update process named ProcName is no longer running scripts to generate view instances. This should only be called when the ProcName view_update process is not running. The statuses of these view instances will be reset to not_generated. After this, the next applicable update_views/4 will try to recreate these view instances.

show_failed(VSDir,VInst) module: view_sys
show_failed(+VSDir,+VInst) displays each failed view instance and consistency view instance, with file information to help a user track down why the generation, or check, of the view failed.

`reset_failed(ViewSys,VInst)` module: view_sys
`reset_failed(+ViewSys,+VInst)` resets view instances with name `VInst` that had failed, i.e., that are marked as `generation_failed`. Their status will be reset to `not_generated`, so after this, the next applicable call to `update_views/4` will try to regenerate the view. If `VInst` is 'all', then views of all instances will be reset.).

`check_viewsys(ViewDir)` module: view_sys
`check_viewsys(+ViewDir)` checks the contents of the `viewsys_view_info.P` file of the `ViewDir` viewsys directory for consistency and completeness.

`viewsys_view_status(+ViewDir,+View:Inst,-Status)` module: view_sys
`viewsys_view_status(+ViewDir,+View:Inst,-Status)` returns the Status of the indicated view in the indicated view instance.

`viewsys_status(+ViewDir)` module: view_sys
`viewsys_status(+ViewDir)` prints out the status of the view system indicated in `ViewDir` for all the options in `viewsys_status/2`.

`viewsys_status(+ViewDir,+Option)` module: view_sys
`viewsys_status(+ViewDir,+Option)` prints out a particular list of view instance statuses as indicated by the value of `option` as follows:

active: View instances currently in the process of being generated.

roots: Root View instances and their current statuses. A root view instance is one that no other view depends on.

failed: View instances whose generation has failed

waiting: View instances whose computations are waiting until views they depend on are successfully update.

checks_waiting: View instances that are waiting for consistency checks to be executed.

checks_failed: View instances whose checks have executed and failed.

`expand_views(ViewSys)` module: expand_views/1
view_sys `expand_views(+ViewSys)` processes view/6 definitions that have a split(N) option, generates the necessary new view/6 facts to do the split, component processing, and rejoin. It overwrites the viewsys_view_info.P file, putting the original view/6 facts into viewOrig/6 facts. This must be called (if necessary) when creating a new viewsys system and before calling generate_view_instance/2.).

`generate_required_dirs(+SubstList,+LogFiles)` module: generate_required_dirs/2
view_sys This predicate can be used to help the user generate `viewsys_required_file/1` facts that may help in configuration and deployment of view systems. It is not needed to create and run normal view systems, only help configure the viewsys_view_info.P

file to support using copy_required_files/2 to move them for deployment, when that is necessary.

generate_required_dirs(+SubstList,+LogFiles) takes an XSB_LOGFILE (or list of XSB_LOGFILEs), normally generated by running a step in the view system, and generates (to userout) viewsys_required_file/1 facts. These can be edited and the copied into the viewsys_view_info.P file to document what directories (XSB code and general data files) are required for running this view system. The viewsys_required_file/1 facts are used by copy_required_files/2 to generate a new set of files that can run the view system.

This predicate can be called in one shell when update_views/4 is running in another shell. This allows the user to monitor the status a long-running invocation of update_views/4.

SubstList is a list of substitutions of the form s(VarString,RootDir) that are applied to @emgeneralize each directory name. For example if we have a large library file structure, in subdirectories of C:/XSBSYS/XSBLIB, the many loaded files (in an XSB_LOGFILE) will start with this prefix, for example,
C:/XSBSYS/XSBLIB/apps/app_1/proc_code.xwam.
By using the substitution, s('DIR','C:/XSBCVS/XSBLIB'), that file name will be abstracted to: 'DIR/apps/app_1' in the viewsys_required_file/1 fact. Then copy_required_files/2 can replace this variable DIR with different roots to determine the source and target of the copying.

LogFiles is an XSB_LOGFILE, that is generated by running xsb and initially calling machine:stat_set_flag(99,1). This will generate a file named XSB_LOGFILE.txt (in the current directory) that contains the names of all files loaded during that execution of xsb. (If the flag is set to @ttK > 1, then the name of the generated file will be XSB_LOGFILE_<K>.txt where <K> is the number K.)

So, for example, after running three steps in a workflow, setting flag 99 to 2, 3, and 4 for each step respectively, one could execute:

```
| ?- generate_required_dirs([s('$DIR$','C:/XSBCVS/XSBLIB')],
                             ['XSB_LOGFILE_2.txt',
                              'XSB_LOGFILE_3.txt',
                              'XSB_LOGFILE_4.txt']).
```

which would print out facts for all directories for files in those LOGFILEs, each with the root directory abstracted.

copy_required_files(+VSDir,+FromToSubs) module: view_sys

This predicate can be used (perhaps with configuration help from generate_required_dirs/2) to copy and deploy view systems and the files they need to run. This predicate is not needed for normal execution of view systems.

copy_required_files(+VSDir,+FromToSubs) uses the viewsys_required_file/1 facts in the viewsys_view_info.P file in the VSDir viewsys directory to copy all directories (and files) in those facts. FromToSubs are terms of the form s(USERVAR,FROMVAL,TOVAL), where USERVAR is a variable in the file templates in the viewsys_required_file/1 facts. A recusrive cp shell command will be generated and executed for each template in viewsys_required_file/1, the source file being the template with USERVAR replaced by FROMVAL and the target File being the template with USERVAR replaced by TOVAL.

All necessary intermediate directories will be automatically created.

E.g.,

```
copy\_required\_files('.',[s('$DIR$','C:/XSBSYS/XSBLIB','C:/XSBSYS/XSBTEST/XSBLIB')
```

would copy all files/directories indicated in the viewsys_required_file/1 facts in the local viewsys_view_info.P file from under C:/XSB/XSBLIB to a (possibly) new directory C:/XSBSYS/XSBTEST/XSBLIB (assuming all file templates were rooted with *DIR*

Chapter 14

The `persistent_tables` Package

By David S Warren

This package supports the generation and maintenance of persistent tables stored in data files on disk (in a choice of formats.) Persistent tables store tuples that are computed answers of subgoals, just as internal XSB tables do. Persistent tables allow tables to be shared among concurrent processes or between related processes over time. XSB programmers can declare a predicate to be persistently tabled, and the system will then, when a subgoal for the predicate is called, look to see if the corresponding table exists on disk, and, if it does, read the tuples that are answers for the subgoal on demand from the data file. If the persistent table for the subgoal does not exist, the XSB subgoal will be called and the tuples that are returned as answers will be stored on disk, and then returned to the call. Persistent tables cannot be recursively self-dependent, unlike internal XSB tables. Normally the tables use call subsumption and abstracted from the original call. They act like (internal) subsumptive tables with call abstraction.

A persistent table can serve to communicate between two XSB processes: a process that requests the evaluation of a subgoal and a sub-process that evaluates that subgoal. This is done by declaring a persistently tabled predicate to have its subgoals be evaluated by a subprocess. In this case, when a persistent table for a subgoal needs to be created, a subprocess will be spawned to compute and save the subgoal answers in the persistent table. The calling process will wait for the table to be computed and filled and, when the table is completed, will continue by reading and returning the tuples from the generated persistent table to the initial calling subgoal.

Persistent tables and internal tables (i.e., normal XSB tables) are independent: a predicate may be persistently tabled but not (internally) tabled, tabled but not persistently tabled, neither or both. In many cases one will want to (internally) table a persistently tabled predicate, but not always.

Persistent tables provide a declarative mechanism for accessing data files, which could

be generated by other mechanisms such as the viewsys package, or by other programming languages or organizations. When this is done, simply invoking the goal will access the persistent table, i.e., the data from the data file. In such a case, the data file format must conform to the format declared for the persistent table for its goal.

14.1 Using Persistent Tables with viewsys

Persistent tables can be used as views in the viewsys package.This is done by:

1. Defining a module that contains persistent tabled predicates that correspond to the desired (stored) views.
2. Using pt_need/1 declarations (see below) to declare table dependencies to support concurrent table evaluation.
3. Running a view-generation process (pt_fill/1/2) to compute the desired views by calling XSB processes. The view-generation process will ""pre""-compute the requred tables in a bottom-up order, using multiple concurrent processes as specified. Since no XSB persistently tabled predicate will be called until after all the persistent tables that it depends on have been computed, all XSB predicates will run using those precomputed persistent tables, without blocking and without having to re-compute any of them.

The persistent_tables subsystem maintains persistent tables in directories and files in a subdirectory of the directory containing the source code for a module that defines persistently tabled predicates. The subdirectory is named xsb_persistent_tables. Only predicates defined in a (non-usermod) module can be persistently tabled. For each module with declared persistent tables, there is a subdirectory (whose name is the module name) of xsb_persistent_tables that contains the contents of its tables. In such a subdirectory there is a file, named PT_Directory.P, that contains information on all existent persistent tables (stored or proposed.) The subdirectory also contains all the files that store the contents of persistent tables for the given module.

Currently the way a predicate is declared to be persistently tabled is somewhat verbose. This is because, at this time, there is no XSB compiler support for persistent tables, and therefore the user must define explicitly all the predicates necessary for the implementation. [1]

The following declarations are needed in any module <Module> that uses persistent tables:

```
:- packaging:bootstrap_package('persistent_tables','persistent_tables').
:- import table_persistent/5, pt_call/1 from persistent_tables.
```

[1] In the future, if this facility proves to be useful, we will extend the compiler to simplify the necessary declarations.

```
:- export ensure_<Module>_loaded/0.
ensure_<Module>_loaded.
```

The ensure_<Module>_loaded/0 predicate is called by the system when it is required that the module be loaded.

A persistent table for predicate Pred/K is declared and defined as follows:

```
:- export <Pred>/K, <Pred>_ptdef/K.
:- table_persistent(PredSkel,ModeList,TableInfo,ProcessSpec,DemandGoal).
PredSkel :- pt_call(PredSkel).
Pred_ptdef(....) :- ... definition of Pred/K ....
```

PredSkel indicates a most-general goal for the predicate Pred/K.

As can be seen, the user must define an auxiliary predicate, in this case <Pred>_ptdef/K. This predicate is defined using the clauses intended to define Pred/K. Pred/K itself is defined by the single clause that calls the persistent-tabling meta-predicate pt_call/1. This meta-predicate will generate subgoals for Pred_undef/K and call them as is required.

The arguments of the table_persistent/5 declaration are as follows:

- PredSkel: is the goal whose instances are to be persistently tabled. Its arguments must be distinct variables.
- ModeList: a list of mode-lists (or a single mode-list.) A mode-list is a list of constants, +, t, -, and -+ with a length equal to the arity of Goal. A - mode indicates that the corresponding position of a call to this goal may be bound or free and is to be abstracted when filling the persistent table; a + mode indicates that the corresponding position must be bound and is not abstracted, and so a separate persistent table will be kept for each call bound to any specific constant in this argument position; a t mode indicates that this argument must be bound to a timestamp value. I.e., it must be bound to an integer obtained from the persistent tabling system that indicates the snapshot of this table to use. (See add_new_table/2 for details on using timestamps.) A -+ mode indicates that the corresponding argument may be bound or free, but on first call, it will be abstracted and a separate table will be constructed for each value that this argument may take on. So it is similar to a - mode in that it is abstracted, but differs in that it generates multiple tables, one for each distinct value this argument takes on. This can be used to split data into separate files to be processed concurrently.

  There may be multiple such mode-lists and the first one that a particular call of Goal matches will be used to determine the table to be generated and persistently stored. A call does *not* match a mode-list if the call has a variable in a position that is a + in that mode-list. If a call does not match any mode-list, an error is thrown. Clearly if any mode list contains a t mode, all must contain one in the same position.

- `TableInfo`: a term that describes the type and format of the persistent tables for this predicate. It currently has only the following possibilities:
  - `canonical`: indicates that the persistent table will be stored in a file as lists of field values in XSB canonical form. These files support answers that contain variables. (Except, answers to goals with modes of `-+` must be ground.)
  - `delimited(OPTS)`: indicates that the persistent table will be stored in a file as delimited fields, where OPTS is a list of options specifying the separator (and other properties) as described as options for the predicate `read_dsv/3` defined in the XSB lib module `proc_files`. Goal answers stored in these files must be ground.
- `ProcessSpec`: a term that describes how the table is to be computed. It can be one of the following forms:
  - `xsb`: indicating that the persistent table will be filled by calling the goal in the current xsb process.
  - `spawn_xsb`: indicating that the persistent table will be filled by spawning an xsb process to evaluate the goal and fill the table.
- `DemandGoal`: a goal that will be called just before the main persistently tabled goal is called to compute and fill a persistent table. The main use of this goal is to invoke `pt_need/1` commands (see below) to indiate that the persistent tables that this goal depends on are needed. This allows tables that will be needed by this computation to be computed concurrently by other processes.

14.2 Methodology for Defining View Systems

As mentioned above, persistent tables can be used to construct view systems, i.e., DAGs representing expressions over functions on relations. A relational function is a basic view definition. An expression over such functions is a view system. The leaf relations in the expression are the base relations, and every sub-expression defines a view. A view expression can be evaluated bottom up, given values for every base relation. Independent subexpressions can be evaluated in parallel. Failing computations can be corrected, and only those views depending on a failed computation need to be re-computed.

Sometimes view systems are required to be ""incremental"". That is, given a completely computed view system, in which the base relations are given and all derived relations have been computed, we are given tuples to add to (and maybe delete from) the given base relations, and we want to compute all the new derived view contents. In many systems such incremental changes to the base relations result in incremental changes to the derived

relations, and those new derived relations can be computed in much less time than would be required to recompute all the derived relations starting from scratch with the new (updated) base relations.

To implement a view system in XSB using persistent tables, each view definition is provided by the definition of a persistently tabled predicate. Then given table instances for the base relations, each view goal can be called to create a persistent table representing the contents of the corresponding derived view.

The following describes, at a high level, a methodology for implementing a given view system in XSB using persistent tables.

1. Define the top-level view relations, just thinking Prolog, in a single XSB module. A top-level relation is the ultimate desired output of a view system, i.e., a relation that is normally not used in the definition of another view. Define supporting relations as seems reasonable. Don't worry about efficiency. Use Prolog intuitions for defining relations. Don't worry about incrementality; just get the semantics defined correctly.

2. Now think about bottom-up evaluation. I.e., we use subsumptive tables, so goals will be called (mostly) open, with variables as arguments. Decide what relations will be stored intermediate views. Restructure if necessary to get reasonable stored views.

3. Now make it so the stored views can be correctly evaluated bottom-up, i.e., with an open call. This will mean that the Prolog intuition of passing bound values downward into called predicates needs to be rethought. For bottom-up evalution, all head variables have to be bound by some call in the body. So some definitions may need new body calls, to provide a binding for variables whose values had been assumed to be passed in by the caller.

4. Declare the stored views as table_persistent, and test on relatively small input data. For each table_persistent, decide initially whether to compute it in the given environment or to spawn a process to evaluate in a new process environment.

5. If you don't need incrementality (i.e., given relatively small additions/deletions to the base relations, compute the new derived relations without recomputing results for old unchanged data): then tune (maybe adding split-compute-join concurrency, using the `-+` mode, as appropriate.) And you're done.

6. If you *do* need incrementality: In principle, the system ought to be able automatically to transform the program given thus far into an incremental version. (See Annie Liu's research.) But at this point, I don't know how to do this ensuring that the reslting performance is close to optimal. (Maybe Annie does, but...) So we will transform the existing program by hand, and we will give ""rules-of-thumb"" to help in this process.

14.3 Using Timestamps (or version numbers)

The persistent table package provides some support for integer timestamps for versioning of tables. The programmer can define view predicates with an argument whose value is a version number. The version number must be bound on all calls to persistently tabled goals that contan them. Normally a subgoal of a persistently tabled predicate with a given version number will depend on other subgoals with the same version. This allows the programmer to keep earlier versions of tables for view systems, in order to back out changes or to keep a history of uses of the view system. So normally a new set of base tables will get a new version number, and then all subgoals depending of those base tables will have that same version number.

The `pt_add_table/3` predicate will add base tables and give them a new version number, returning that new version number. This allows the programmer to use that version number in subsequent calls to `pt_fill` to fill the tables with the correct version. Also, when calling the predicate `pt_eval_viewsys/5` the `Time` variable can be used in the subgoals in the `FillList` to invoke the correctly versioned subgoals.

A particularly interestng use of versions is in the implementation of incremental view systems. Recall that in an incremental view system, one has a table that contains the accumulated records named, say, old_records/5, and receives a base table of new records to process named, say, new_records/5. The incremental view system will define an updated record file named, say, all_records/5, which will contain the updated records after processing and includng the new_records. It is natural to use versions here, and make each predicate old_record/5, new_record/5, and old_record/5 have a version argument, say the first argument. Then note that we can define old_records in terms of the previous version of all_records, as follows:

```
old_records(Time,....) :-
     Time > 1,
     PrevTime is Time - 1,
     all_records(PrevTime,...).
```

Note that the version numbers, being always bound on call (and treated according to a `+` mode), will not appear in any stored table. The numbers will appear only in the called subgoals that are stored in the `table_instance/8` predicate in the `PT_Directory.P` file. So using version numbers does not make the persistent tables any larger.

14.4 Predicates for Persistent Tabling

`pt_call(+Goal)` `module: pt_call/1`

persistent_tables This predicate assumes that `Goal` is persistently tabled and calls it.

This predicate is normally used only in the definition of the `_ptdef` version of the persistently tabled predicate, as described above.

If the table for `Goal` exists, it reads the table file and returns its answers. If the table file is being generated, it waits until it is generated and then reads and returns its answers. If the table file doesn't exist and is not in the process of being generated, it generates the table and then returns its results. If the persistent table process declaration indicates `spawn_xsb`, it spawns a process to generate the table and reads and returns those answers when the process is completed. If the process indication is `xsb`, it calls the goal and fills the table if necessary, and returns the answers.

`pt_fill(+GoalList)` `module: pt_fill/1`

persistent_tables The predicate `pt_fill(+GoalList)` checks if the persistent table for each persistently tabled `Goal` in `GoalList` exists and creates it if not. It should always succeed (once, unless it throws an error) and the table will then exist. If the desired table is already generated, it immediately succeeds. If the desired table is being generated, it looks to see if there is another table that is marked as `needs_generating` and, if so, invokes the `pt_fill/1` operation for that table. It continues this until it finds that `Goal` is marked as `generated`, at which time it returns successfully. If no table for `Goal` exists or is being generated, it generates it.

`pt_fill(+Goal,+NumProcs)` `module: pt_fill/2`

persistent_tables `pt_fill(+Goal,+NumProcs)` is similar to `pt_fill/1` except that it starts `NumProcs` processes to ensure that the table for `Goal` is generated. Note that filling the table for `Goal` may require filling many other tables. And those table may become marked as `needs_generation`, in which case multiple processes can work concurrently to fill the required tables.

`pt_need(+Goals)` `module: pt_need/1`

persistent_tables `pt_need(+Goals)` creates table entries in the `PT_Directory.P` file for each persistently tabled `Goal` in the list of goals `Goals`. (`Goals` alternatively may be a single persistently tabled goal. The new entry is given status `needs_generation`. This predicate is intended to be used in a goal that appears as the 5th argument of a `table_persistent/5` declaration. It is used to indicate other goals that are required for the computation of the goal in the first argument of its `table_persistent/5` declaration. By marking them as ""needed"", other processes (started by a call to `pt_fill/2`) can begin computing them concurrently. Note that these `Goals` can share variables with the main `Goal` of the declaration, and thus appropriate instances of the subgoals can be generated. For example, if time stamps are used, the needed subgoals should have the same variable as the main goal in the corresponding ""time"" positions.

Note that a call to `pt_need/1` should appear **only** in the final argument of a `table_persistent/5` declaration. Its correct execution requires a lock to be held and predicates to be loaded, which are ensured when that goal is called, but cannot be correctly ensured by any other call(s) to the `persistent_tables` subsystem.

table_persistent(+Goal,+Modes,+TableInfo,+ProcessSpec,+DemandGoal) module: table_persis
persistent_tables This predicate (used as a directive) declares a predicate to be persistently tabled. The form is table_persistent(+Goal, +Modes, +TableInfo, +ProcessSpec, +DemandGoal), where:

- Goal: is the goal whose instances are to be persistently tabled. Its arguments must be distinct variables. Goal must be defined by the single clause:

  ```
  Goal :- pt_fill(Goal).
  ```

  Clauses to define the tuples of Goal must be associated with another predicate (of the same arity), whose name is obtained from Goal's predicate name by appending _ptdef.

- ModeList: a list of mode-lists (or a single mode-list.) A mode-list is a list of constants, +, t, -, and -+ with a length equal to the arity of Goal. The mode indicates puts constraints on the state of corresponding argument in a subgoal call. A ""-"" mode indicates that the corresponding position of the goal is to be abstracted for the persistent table; a ""+"" mode indicates that the corresponding position is not abstracted and a separate persistent table will be kept for each call bound to any specific constant in this argument position; a ""t"" mode indicates that this argument will have a ""timestamp"". I.e., it will be bound to an integer obtained from the persistent tabling system that indicates the snapshot of this table to use. (See add_new_table/2 for details on using timestamps.) A mode of ""-+"" is similar to a ""-"" mode in that the associated argument is abstracted. The difference is that instead of all the answers being stored in a single table, there are multiple tables, one for each value of this argument for which there are answers.

  There may be multiple such mode-lists and the first one that a particular call of Goal matches will be used to determine the table to be generated and persistently stored. A call does *not* match a mode-list if the call has a variable in a position that is a ""+"" in that mode-list. If a call does not match any mode-list, an error is thrown. If any mode list contains a t mode, all must contain one in the same position.

- TableInfo is a term that describes the type and format of the persistent tables for this predicate. It may have the following forms, with the described meanings:
  - file(canonical): indicates that the persistent table will be stored in a file as lists of field values in XSB canonical form.
  - file(delimited(OPTS)): indicates that the persistent table will be stored in a file as delimited fields, where OPTS is a list of options specifying the separator (and other properties) as described as options for the predicate read_dsv/3 in the XSB lib module proc_files.

- ProcessSpec is a term that describes how the table is to be computed. It can be one of the following forms:
  - xsb: indicating that the persistent table will be filled by calling the goal in the current xsb process.
  - spawn_xsb: indicating that the persistent table will be filled by spawning an xsb process to evaluate the goal and fill the table.
- DemandGoal: a goal that will be called just before the main persistently tabled goal is called to compute and fill a persistent table. The main use of this goal is to invoke pt_need/1 commands (see below) to indicate to the system that the persistent tables that this goal depends on are indeed needed. This allows tables that will be needed by this computation to be computed by other processes. This is the way that parallel computation of a complex query is supported.

pt_abolish_subgoals(+GoalList) module: pt_abolish_subgoals/1
persistent_tables pt_abolish_subgoals(+GoalList) abolishes the persistent tables for all goals in GoalList by removing the corresponding facts in table_instance. The table files containing the data remain, and can be cleaned up using pt_remove_unused_tables/1.

pt_move_tables(+MoveList) module: pt_move_tables/1
persistent_tables pt_move_tables(+MoveList) moves persistent tables. MoveList is a list of pairs of goals of the form FromGoal > ToGoal, where FromGoal and ToGoal are persistently tabled goals and their persistent tables have been filled. For each such pair the table file for ToGoal is set to the file containing the table for FromGoal. The table files must be of the same format. FromGoal has its table_instance fact removed. This predicate may be useful for updating new and old tables when implementing incremental view systems.

pt_remove_unused_tables(+Module) module: pt_remove_unused_tables/1
persistent_tables This predicate cleans up unused files from the directory that stores persistent tables. pt_remove_unused_tables(+Module) looks through the PT_Directory.P file for the indicated module and removes all files with names of the form (table_<Tid>.P) (or .txt) for which there is no table id of <Tid>. So a user may delete (or abolish) a persistent table by simply editing the PT_Directory.P file (when no one is using it!) and deleting its table_instance fact. Then periodically running this predicate will clean up the storage for unnecessary tables.

pt_reset(+Module) module: pt_reset/1
persistent_tables pt_reset(+Module) processes the PT_Directory.P file and deletes all table_instance records for tables that have status being_generated. This will cause them to be re-evaluated when necessary. This is appropriate to call if all processes computing these tables have been aborted and were not able to update the directory. It may also be useful if for some reason all processes are waiting for something to be done and no progress is being made.

`pt_delete_later(Module,TimeStamp)` `module: pt_delete_later/2`

persistent_tables `pt_delete_later(Module,TimeStamp)` delete all tables that have a timestamp larger than `Timestamp`. It keeps the tables of the TimeStamp snapshot. It deletes the corresponding table records from the PT_Directory, and removes the corresponding files that store the tuples.

`pt_delete_earlier(Module,TimeStamp)` `module: pt_delete_earlier/2`

persistent_tables `pt_delete_earlier(Module,TimeStamp)` delete all tables that have a timestamp smaller than `Timestamp`. It keeps the tables of the TimeStamp snapshot. It deletes the corresponding table records from the PT_Directory, and removes the corresponding files that store the tuples.

`pt_delete_table(+Goal)` `module: pt_delete_table/1`

persistent_tables `pt_delete_table(+Goal)` deletes the table for `Goal` in its `PT_Directory.P` file, so it will need to be regenerated when next invoked. The actual file containing the table data is *not* removed. (It may be a file in another directory that defines the table via a call to `pt_add_table/2` or friend.) To remove a local file that contains the tabled data, use `pt_remove_unused_tables/1`.

`pt_add_table(+Goal,+FileName)` `module: pt_add_table/2`

persistent_tables `pt_add_table(+Goal,+FileName)` uses the file `FileName` to create a persistent table for `Goal`. `Goal` must be persistently tabled. It creates a new table_instance record in the `PT_Directory.P` file and points it to the given file. The file is not checked for having a format consistent with that declared for the persistently tabled predicate, i.e., that it is correctly formated to represent the desired tuples. The user is responsible for ensuring this.

pt_add_table(Goal0,FileName) :- pt_add_table(Goal0,FileName,none).

`pt_add_table(+Goal,+FileName,?TimeStamp)` `module: pt_add_table/3`

persistent_tables `pt_add_table(+Goal,+FileName,?TimeStamp)` uses the file `FileName` to create a persistent table for `Goal`, which must be persistently tabled. It returns in `TimeStamp` a new (the next) time stamp for this module (obtained from the fact for predicate `table_instance_cnt/2` in the ET Directory.) It is assumed that `Goal` has a time argument and the returned value will be used in its eventual call.

This predicate creates a new table_instance record in the `PT_Directory.P` file and sets its defining file to be the value of `FileName`. The file is not checked for consistency, that it is correctly formated to represent the desired tuples. The user is responsible for insuring this.

`pt_add_tables/2` `module: pt_add_tables(+GoalList,+FileList)`

persistent_tables `pt_add_tables(+GoalList,+FileList)` is similar to `pt_add_table/2` but takes a list of goals and a corresponding list of files, and defines the tables of the goals using the files.

`pt_add_tables(+GoalList,+FileList,-Time)` `module: pt_add_tables/3`
persistent_tables `pt_add_tables(+GoalList,+FileList,-Time)` is similar to `pt_add_table/3` but takes a list of goals and a corresponding list of files, and defines the tables of the goals using the files, returning the snapshot time in `Time`.

`pt_eval_viewsys(+GoalList,+FileList,-Time,+FillList,+NProcs)` `module: pt_eval_viewsys/5`
persistent_tables The predicate `pt_eval_viewsys(+GoalList, +FileList, -Time, +FillList, +NProcs)` adds user files containing base tables to a persistent tabling system and invokes the computing and filling of dependent tables. `GoalList` is a list of subgoals that correspond to the base tables of the view system. `FileList` is the corresponding list of files that contain the data for the base tables. They must be formated as the `table_persistent` declarations of their corresponding subgoals specify. `Time` is a variable that will be set to the timestamp, if the base goals of `GoalList` contain time stamp arguments. `FillList` is a list of persistently tabled subgoals to be filled (using `pt_fill/1/2`.) `NProcs` is an integer indicating the maximum number of processes to use to evaluate the view system. This predicate provides a simple interface to `pt_add_tables/3` and `pt_fill/2`.

Chapter 15

PITA: Probabilistic Inference

By Fabrizio Riguzzi

Probabilistic Inference with Tabling and Answer subsumption (PITA) [11, 10] is a package for reasoning under uncertainty. In particular, PITA supports various forms zof Probabilistic Logic Programming (PLP) and Possibilistic Logic Programming (PossLP). It accepts the language of Logic Programs with Annotated Disjunctions (LPADs)[20, 21] and CP-logic programs [18, 19].

An example of LPAD/CP-logic program is as follows (the syntax in the PITA implementation is slightly different, as explained in Section 15.2)

$$\begin{aligned}(heads(Coin):0.5) \vee (tails(Coin):0.5) &\leftarrow toss(Coin), \neg biased(Coin).\\ (heads(Coin):0.6) \vee (tails(Coin):0.4) &\leftarrow toss(Coin), biased(Coin).\\ (fair(Coin):0.9) \vee (biased(Coin):0.1).&\\ toss(Coin).&\end{aligned}$$

The first clause states that if we toss a coin that is not biased it has equal probability of landing heads and tails. The second states that if the coin is biased it has a slightly higher probability of landing heads. The third states that the coin is fair with probability 0.9 and biased with probability 0.1 and the last clause states that we toss a coin with certainty.

PITA computes the probability of queries by tranforming the input program into a normal logic program and then calling a modified version of the query on the transformed program. In order to combine probabilities or possibilities from different derivations of a goal, PITA makes use of tabled answer subsumption. For PLPs, PITA's answer subsumption makes use of the BDD package CUDD to combine the possibly non-independent probabilities of different derivations. CUDD is included in the XSB distribution.

15.1 Installation

To install PITA with XSB, run XSB `configure` in the `build` directory with option `-with-pita` and then run `makexsb` as usual. On most Linux systems, this is all that is needed.

- *Windows* When compiling in cygwin, also build the cygwin dll with `makexsb cygdll`.
- *MacOS* When compiling on MacOS, it should be noted that recent versions of `xcode` do not include `autoconf` and `automake`, both of which are needed for the PITA installation. If your these tools are not installed on your system, they can be easily installed by

  ```
  (sudo) brew install autoconf
  (sudo) brew install automake
  ```

  or

  ```
  sudo port autoconf
  sudo port automake
  ```

15.2 Syntax

Disjunction in the head is represented with a semicolon and atoms in the head are separated from probabilities by a colon. For the rest, the usual syntax of Prolog is used. For example, the CP-logic clause

$$h_1 : p_1 \vee \ldots \vee h_n : p_n \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_l$$

is represented by

```
h1:p1 ; ... ; hn:pn :- b1,...,bm,\+ c1,....,\+ cl
```

No parentheses are necessary. The `pi` are numeric expressions. It is up to the user to ensure that the numeric expressions are legal, i.e. that they sum up to less than one. Other points about `pita` syntax are:

- If the clause has an empty body, it can be represented as:

  ```
  h1:p1 ; ... ;hn:pn.
  ```

- If the clause has a single head with probability 1, the annotation can be omitted and the clause takes the form of a normal prolog clause, i.e.

```
h1:- b1,...,bm,\+ c1,...,\+ cl.
```

stands for

```
h1:1 :- b1,...,bm,\+ c1,...,\+ cl.
```

- The probabilities in the heads may sum to a number less than 1. For instance, the LPAD clause

$$h_1 : p_1 \vee null : (1 - p_1) \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_l$$

is represented in `pita` by dropping the `null` conjunct, i.e.,

```
 h_1:p_1 :-  b_1,\dots,b_m ,\+ c_1,\ldots,\+ c_l.
```

- Finally, the body of clauses can contain a number of built-in predicates including:

```
is/2 >/2 </2 >=/2 =</2 =:=/2 =\=/2 true/0 false/0
=/2 ==/2 \=/2 \==/2 length/2 member/2
```

The directory `$XSB_DIR/packages/pita/examples` contains several examples of LPADs, including the program `coin.cpl` above, which is written in PITA's syntax as:

```
heads(Coin):1/2 ; tails(Coin):1/2:-
     toss(Coin),\+biased(Coin).
heads(Coin):0.6 ; tails(Coin):0.4:-
     toss(Coin),biased(Coin).
fair(Coin):0.9 ; biased(Coin):0.1.
toss(coin).
```

15.3 Using PITA

15.3.1 Probabilistic Logic Programming

PITA accepts input programs in two formats: `.cpl` and `.pl`. In both cases they are translated into an internal form that has extension `.P`. In the `.cpl` format, files consist of a sequence of LPAD clauses. In the `.pl` format, files use the syntax of `cplint` for SWI-Prolog, see http://friguzzi.github.io/cplint/_build/html/index.html. In the `.pl` format, the same file can be used for PITA in XSB and PITA in `cplint` for SWI-Prolog.

If you want to use inference on LPADs load PITA in XSB with

```
?- [pita].
```

Then you have different commands for loading the input file.

If the input file is in the `.cpl` format, you can translate it into the internal representation and load it with

```
?- load_cpl(coin).
```

Note that `coin.cpl`, which is not in Prolog syntax **cannot** be loaded via the normal command to compile and load a Prolog file (`?- [coin]`).

This commands reads `coin.cpl`, translates it into `coin.cpl.P` and loads `coin.cpl.P`.

For files in the `.pl` format, the command is

```
?- load_pl(coin).
```

that reads `coin.pl`, translates it into `coin.pl.P` and loads `coin.pl.P`.

You can also use command

```
?- load('coin.pl').
```

that requires the full file name, including the extension, compiles it into a file with the same name with the added extension `.P` and loads it.

You can also load directly the translated (compiled) version of a file with the command

```
?- load_comp('coin.cpl.P').
```

of

```
?- load_comp('coin.pl.P').
```

that loads directly the compiled file.

Next, the probability of query atom `heads(coin)` can be computed by

```
?- prob(heads(coin),P).
```

PITA, which is based on the distribution semantics (cf. [15]) will give the answer `P = 0.51` to this query.

The package also includes a test file that can be run to check that the installation was successful. The file is `testpita.pl` and it can be loaded and run with

```
?- [testpita].
?- test_pita.
```

The package also includes MCINTYRE, which performs approximate inference with Monte Carlo algorithms. MCINTYRE accepts the same input formats as PITA and the same commands for loading input files. See http://friguzzi.github.io/cplint/_build/html/index.html for a description of the available commands.

For loading MCINTYRE use

```
?- [mcintyre].
```

File `testmc.pl` can be used for testing MCINTYRE. The command to run the tests is

```
?- [testmc].
?- test_mc.
```

The `examples` folder contains various examples of use of MCINTYRE. You can also look at the file `test_mc.pl` for a list of example and queries over them.

The package also includes SLIPCOVER, an algorithm for learning LPADs. Input files should follow the syntax specified in http://friguzzi.github.io/cplint/_build/html/index.html and should have the `.pl` extension. They can loaded for example with

```
?- load_pl(bongard).
```

for `bongard.pl`.

File `testsc.pl` can be used for testing SLIPCOVER. The command to run the tests is

```
?- [testsc].
?- test_sc.
```

The `examples/learning` folder contains various examples of use of SLIPCOVER. You can also look at the file `test_sc.pl` for a list of example and goals.

15.3.2 Modeling Assumptions

The probability of `heads(coin)` above is calculated by adding the probability of the *composite choices*

$$head(coin), fair(coin) = 0.45$$

and

$$head(coin), biased(coin) = 0.06$$

These two composite choices are mutually exclusive since they differ in their atomic choices (in this case, the atoms `fair(coin)` and `biased(coin)`). Accordingly, their probabilities can be added leading the total 0.51. More about the theory that underlies the distribution semantics can be found in the survey article [12].

In the above discussion of the coin example, we combined probabilities according to the full distribution semantics. However, some programs may satisfy a set of modeling assumptions that allows programs to be evaluated much more efficiently.

- *The independence assumption*: The assumption that different calls to a probabilistic atom can we be evaluated independently. This leads to the ability to compute the probability of a conjunction (A, B) as the product of the probabilities of A and B;
- *The exclusiveness assumption* The assumption that different derivations of an atom A depend on exclusive composite choices. This leads to the ability to compute the probability of an atom as the sum of the probabilities of its derivations.

While these assumptions are in fact satisfied by the `coin` program, they may be fairly strong for larger programs.

These assumptions are fairly strong – note that the `coin` program discussed above does not satisfy the exclusiveness assumption, since the two derivations of `head(coins)` share the probabilistic atom , as used for instance in the PRISM system [16], i.e.:

Example 15.3.1 An example of a program that does not satisfy the exclusiveness assumption is `$XSB_DIR/packages/pita/examples/flu.cpl`

```
sneezing(X):0.7 :- flu(X).
sneezing(X):0.8 :- hay_fever(X).
flu(bob).
hay_fever(bob).
```

Given the query `sneezing(bob)`, four possible total composite choices or *worlds* must be considered.

Clause 1	`sneezing(bob)`	`sneezing(bob)`	*null*	*null*
Clause 2	`sneezing(bob)`	*null*	`sneezing(bob)`	*null*
Probability	0.56	0.14	0.24	0.06

Note that unlike in the `coin` program, the derivations of `sneezing(bob)` in the first two clauses are not mutually exclusive; rather they need to be expanded into mutually exclusive worlds, and the probabilities of those worlds in which `sneezing(bob)` is true can then be summed. In this case, probability of `sneezing(bob)` is the probability of all worlds in which `sneezing(bob)` is true, which is $0.56 + 0.14 + 0.24 = 0.94$.

If you know that your program satisfies the independence and exclusion axioms, you can perform faster inference with the PITA package `pitaindexc.P`, which accepts the same commands of `pita.P`. Due to its assumptions, it does not need to maintain information about composite choices in the CUDD BDD system [1]

If you want to compute the Viterbi path and probability of a query (the Viterbi path is the explanation with the highest probability) as with the predicate `viterbif/3` of PRISM, you can use package `pitavitind.P`.

The package `pitacount.P` can be used to count the explanations for a query, provided that the independence assumption holds. To count the number of explanations for a query use

```
:- count(heads(coin),C).
```

`pitacount.P` does not need to maintain composite choices as BDDs in Cudd, and so can be much faster than computing the full distribution semantics, or the Viturbi path.

15.3.3 Possibilistic Logic Programming

PITA can be used also for answering queries to possibilistic logic program [2], a form of logic progamming based on possibilistic logic [3]. The package `pitaposs.P` provides possibilistic inference. You have to write the possibilistic program as an LPAD in which the rules have a single head whose annotation is the lower bound on the necessity of the clauses. To compute the highest lower bound on the necessity of a query use

```
:- poss(heads(coin),P).
```

Like `pitaindexc` and `pitacount`, `pitaposs` does not require maintenance of composite choices through BDDs in CUDD.

[1] Computing the full distribution semantics for a ground program P is $\#P$-complete, while computing the restricted distribution semantics has the same low polynomial complexity as computing the well-founded semantics: $\mathcal{O}(size(P) \times atoms(P))$.

Chapter 16

minizinc: The XSB Interface to MiniZinc-based Constraint Solving

By Michael Kifer

16.1 Introduction

MiniZinc is a uniform declarative constraint language that is understood by most modern solvers for constraint and optimization problems. It comes bundled with a few such solvers, one of which, `gecode`, is top-notch: very powerful and fast. Other solvers, including most of the newest ones, can be downloaded separately and installed as plugins.

The MiniZinc language is described in https://www.minizinc.org/doc-2.2.3/en/index.html; see, especially, the tutorial. The XSB interface to MiniZinc comes with several sample problems from that tutorial, which are found in `.../XSB/packages/minizinc/examples/`. The file `.../XSB/packages/minizinc/examples.P` contains examples of XSB invocations of those problems. After the installation, all examples can be run by simply starting XSB and then

```
| ?- [minizinc].   %% load the package
| ?- [examples].   %% run all examples
```

16.2 Installation

Some Linux distributions (e.g., Ubuntu) come with ready-made MiniZinc .deb or .rpm packages. However, one must make sure that the command `minizinc` is provided by those packages (some provide only the IDE). Mac packages are also available.

In case a Linux or a Mac package is incomplete (or if one uses Windows), MiniZinc can be downloaded from https://www.minizinc.org/software.html.

Once installed, make sure that the command `minizinc` is understood when typed in a command window. If not, add *folder-to-where-*/**bin**/minizinc is sitting to the environment variable `PATH`. In Windows, this is best done in Control Panel; in Linux and Mac, add the command

```
export PATH=$PATH:path-to-minizinc
```

to `.bashrc` or an equivalent place. For instance,

```
PATH=$PATH:$HOME/minizinc/MiniZincIDE-2.2.3-bundle-linux/bin
```

16.3 The API

Constraint and optimization problems are specified using the MiniZinc language in *model files*, which have the suffix `.mzn`. Such problems usually have a number of *input variables* and several *output* (or *decision*) *variables*. In principle, input values can be specified in the model file itself, but this is generally not a good idea because typically one wants to solve the same problem with different inputs. For this reason, MiniZinc allows one to use models (that have no input) with one or more *input files*, which have the extension `.dzn`. In addition, XSB's API to MiniZinc lets one pass parameters in-line, as part of a Prolog call.

Important: the MiniZinc model files (.mzn) must **not** have `output` statements in them. Otherwise, errors and wrong answers may result. (These output statements will be added automatically based on output templates described below.)

The API itself mainly consists of the following calls, which live in XSB's module `minizinc`:

- `solve(+MznF,+DatFs,+InPars,+Solver,+Solns,+OutTempl,-Rslt,-Xceptns)`: The meaning of the arguments is as follows:
  - `MznF`: should be bound to a path to the desired model file (.mzn file) that contains a specification of a constraint or optimization problem. The path must be represented as a Prolog atom and can be absolute or relative to the current directory.
  - `DatFs`: a list of paths (relative or absolute) to the data files that describe all or some of the input parameters for the model in `MznF`. All paths must be atoms. If no data files are needed, just use the empty list `[]`.
  - `InPars`: a list of the *in-line* input parameters to the model. These are supported for flexibility, to allow initialization of some or all input parameters directly in

Prolog. Each parameter in `InPars` must have the form *id* = *value*, where *id* must be the Id of an input variable used in the model file `MznF` and value must be a term understood by MiniZinc. Typically, such a term would be a number, an atom, or a function term. For instance, `foo='[|1, 0|3, 4|8, 9|]'` would initialize the input variable `foo` with a 2-dimensional array of numbers; `'Item' = anon_enum(8)` would initialize the MiniZinc variable `Item` of an enumerated type to a set `{Item_1,Item_2,...,Item_8}`. Note that `Item` must be quoted on the Prolog side, since otherwise it would be interpreted as a variable.

– `Solver`: the name of the solver to use. If this is a variable, the default solver `gecode` is used.

– `Solns`: the number of solutions to show. Could be `all` or a positive integer. Note: for optimization problems where a function is maximized or minimized, `Solns` must be bound to 1. Otherwise, solvers would return also non-optimal solutions, and the desired optimal one may not be the first.

– `OutTempl`: the template showing how the output should look like. It has the form *predname*(*OutSpec*$_1$, ...,*OutSpec*$_n$) where *predname* is the name of a predicate where the results will be stored (see below) and each *OutSpec* can have one of these forms:

  * An *atom* representing an output ("decision") variable defined in the model file `MznF` or an atom representing an arithmetic expression (in the syntax of MiniZinc, which is close to XSB) involving one or more decision variables. In the result, this atom will be replaced with the value of that variable or expression. Example: 'P*100'.
  * A simple arithmetic expression involving decision variables, numbers, and +, −, *, /. Example: p*100+9. The difference with the above is that the single quotes are omitted, for convenience. Note that if the MiniZinc variable name were P then it would have to be quoted—to protect it from Prolog: 'P'*100+9.
  * A term of the form `+`(*atom*) or `str`(*atom*). In the result, this will be replaced with *atom* verbatim. Note: if `atom` is not alphanumerical or if it starts with a capital letter, it must be quoted.
  * A list of the form [*OutSpec*$_1$, ...,*OutSpec*$_n$]. Each *OutSpec* in the list has the form described here.
  * A term of the form *OutSpec*$_1$ `=` *OutSpec*$_2$. Each *OutSpec* has the form described here.

  The template predicate cannot be `solve/8`, `solve_flex/8`, `show/1`, `delete/1`, and the arity must be greater than 0.

– `Rslt`: must be a term that matches the output template—typically just a variable. Solutions to the constraint and optimization problems will be returned as bindings to variables in this term. The term cannot match the predicates `solve/8`,

`solve_flex/8`, `show/1`, or `delete/1` (e.g., cannot be `solve(_,_,_,_,_,_,_,_)`), and the arity must be greater than 0 (i.e., the predicate cannot be `foo/0` and the like).

Note: solutions are also asserted in the predicate `minizinc:`*SolutionPred*`/`*Arity*, where `SolutionPred/Arity` is the predicate name and arity used in the aforementioned `OutTempl`. Thus, solution predicates from different runs of MiniZinc are accumulated in module `minizinc` and can be used at a later analysis stage. If any of these predicates are no longer needed, they can be emptied out with `retractall/1` or `abolish/1`. For instance, `abolish(minizinc:somepred/5)`.

– `Xceptns`: a list of exceptions returned by the solver. If the constraint/optimization problem was solved successfully, this variable will be bound to an empty list `[]`. Otherwise, each exception has the form `(reason=...,model_file=...)` and the following reasons might be returned:

  * `unsatisfiable` — the optimization problem is unsatisfiable.
  * `unbounded` — the optimization problem has an unbounded objective function. For example, if the problem is to maximize the function and the function has no maximum (under the given constraints); or the problem is to minimize the function, and there is no minimum.
  * `unsatisfiable_or_unbounded` — one of the above.
  * `unknown` — could not find a solution within the limits (e.g., timeout).
  * `error` — search resulted in an error.

Note: exceptions are separate from other kinds of errors, such as a syntax errors in a model or data file or using a solver with a feature or option it does not support. Errors are explained later.

- `solve_flex(+MznF,+DatFs,+InPars,+Solver,+Solns,+OutTempl,-Rslt,-Xceptns)`: The meaning of the parameters is the same as for `solve/8`. The difference is that if `InPars` or `OutTempl` is non-ground, the call to MiniZinc is delayed until they both groun. If the top query finishes and `InPars` or `OutTempl` is still not ground, MiniZinc will *not* be called at all. This is used in cases when it is hard to estimate when `InPars` or `OutTempl` may become ground, so calls to `solve_flex` can be placed early. But, of course, one must ensure that `InPars`/`OutTempl` will get bound at some point.

How does MiniZinc return complex structures back to Prolog? MiniZinc has a number of data structures that do not have direct equivalence in Prolog, so they are mapped to Prolog terms when results are returned as bindings for the `Rslt` variable. Here is the correspondence:

- Numbers are passed back to Prolog as integers and floats.
- MiniZinc strings and identifiers are passed to Prolog as atoms.

- Arrays are returned as lists. Multi-dimensional arrays are flattened and passed as lists as well. For instance, a 2-dimensional array `[|1, 2|3, 4|5, 6|]` will be returned as `[1,2,3,4,5,6]`.
- Sets are returned as terms of the form `{elt1,elt2,elt3,...}`. For example, the set `{a,b,c}` comes back as the term `{a,b,c}`. Note that in Prolog this term is really `'{}'((a,b,c))`. Observe the double parentheses, which indicate that the functor symbol here is `{}/1`, not `{}/3`.
- A MiniZinc range expressions of the form `N1..N2` is returned as `'..'(N1,N2)`. For instance, the range `3..17` comes back to Prolog as `'..'(3,17)`.

Errors vs. exceptions. If a model or data file contains a syntax error or a feature that the chosen solver does not support, the `solve/8` and `solve_flex/8` calls will fail and a message will be printed to standard output:

```
+++ xsb2mzn: syntax or type errors found; details in ....some file...
```

The user will then be able to find the details about the problem.

Note: errors are different from exceptions. If only exceptions are returned (and no errors), the calls `solve/8` and `solve_flex/8` will succeed. In contrast, they will fail in case of errors.

Debugging API. For further development and bug reporting, the following calls are useful. They are all 0-ary predicates that take no arguments; they all reside in the XSB module `minizinc`.

- `keep_tmpfiles`: The MiniZinc interface creates a number of temporary files, which are deleted, if MiniZinc finished normally and without an error. However, if a bug is suspected, it is desirable to preserve these files and send them to the developers. This can be achieved by executing the `keep_tmpfiles` predicate as a query, before the call to `solve/8`.
- `show_mzn_cmd`: Executing this as a query will cause the `solve/8` predicate, described earlier, to print the shell command that was used to invoke MiniZinc in each call. This is useful if one suspects a bug in the API.
- `dbg_clear`: Executing this clears out the flags set by the above debugging calls. As a result, the temporary files will again be deleted after each invocation of MiniZinc and shell commands will not be shown.

Chapter 17

xsbpy: The XSB-Python Interface

Version 0.3

By Muthukumar Suresh, Theresa Swift, Carl Andersen

This chapter provides rough draft documentation of alpha-level software with rapidly changing features.

Although it is still under development the `xsbpy` package already provides an efficient and easy way for XSB to call Python functions and methods. `xsbpy` leverages the fact that XSB and most Pythons are written in C, so that both systems are loaded into the same process. The core interface routines are also written entirely in C, so the interface is very efficient and – it is hoped – very robust within its known limitations.

This chapter first describes how to configure `xsbpy`, followed by introductory examples. Next is a more precise description of its functions, its current limitations followed by some sample applications and further examples.

17.1 Configuration and Loading

An initial version of a configuration process of `xsbpy` has been written for Linux and Windows, and is still in the process of being tested under various confiugrations. On these platforms, `xsbpy` has been tested using versions 3.7 and 3.8 Python libraries.

17.1.1 Configuring `xsbpy` under Linux with GCC

In principle when XSB is configured on Linux, `xsbpy` will also be configured so that it can be used like any other XSB module. However, for this configuration to work several things must be in place.

- GCC (or some other compiler) must be present and usable by the userid performing the cofiguration.
- A development version of Python 3.X must also be present and usable by the userid performing the configuration. This version must be a CPython version (this is the usual implementation of Python), In additiion the version of Python must also contain a version of Python as a shared object (`.so`) file. This often requires an additional download of Python. On Ubuntu, while Python is ordinarily downloaded as:

  ```
  apt-get install python3.x
  ```

  the shared-object library must also be downloaded as, e.g.:

  ```
  apt-get install libpython3.x-dev
  ```

- Next, the user must have permissions to use `pip` on this version of Python, as the configuration process may need to download the Python library `find_libpython`.

If there are difficulties in configuring `xsbpy`, it is likely that one of the above requirements is not met. However, if these requirements are met, but the configuration dows not work properly, it is best to check the file `xsb2py_connect_defs.h` to see whether its definitions are correct. The definitions in this file are used in `get_compiler_options/2` in `init_xsbpy.P` to properly compile `xsbpy` and link in `libpython3x.so`. [1]

In Ubuntu Linux, this is all done automatically when `xsbpy` is consulted into XSB via a normal consult or `ensure_loaded/[1,2]`. As part of this process, in `init_xsbpy.P` the main C code for `xsbpy` is compiled using gcc compiler options that are appropriate for the version of libpython used. [2] When the `xsbpy` module is consulted into XSB, the `libpython` shared object file (or DLL) is loaded dynamically along with the `xsbpy` shared object file.

17.1.2 Configuring `xsbpy` under Windows

17.1.3 Configuring `xsbpy` under MacosX

17.1.4 Testing whether the `xsbpy` configuration was successful

No matter what platform `xsbpy` is installed on, to test out whether `xsbpy` has been loaded and is working properly, simply change the working directory to `xsbpy/test` and execute the command

```
bash test.sh
```

[1] If you can determine that the required software is present and that there is a problem with the configuration please report the problem at `https://sourceforge.net/p/xsb/bugs`.

[2] See `get_compiler_options/2` in `init_xsbpy.P`.

17.2 Introductory Examples

We introduce some of the core functionality of `xsbpy` via a series of simple examples. As background, when the goal `?- [xsbpy].` is executed, Python is loaded and initialized within the XSB process, the core Prolog modules of `xsbpy` is loaded into XSB, and paths to `xsbpy` and its subdirectories are added both to Prolog and to Python. Apart from these directories (added by modifying Python's `sys,path`) XSB calls Python, Python will search for modules and packages in the same manner as if it were stand-alone.

The translation of JSON through xsbpy, although it is functional, is mainly presented for pedagogic purposes. In general, we recommend using XSB's native JSON interface described in Chapter 19 of this manual.

Example 17.2.1 Calling a Python Function (I)

Consider the following call:

```
pyfunc(json,
       loads('{"name": "Bob", "languages": ["English","French","GERMAN"]}'),
       Ret)
```

which loads the Python `json` module if needed, and calls the Python function

```
xsbpy_json.loads()
```

on the above string which converts a JSON string to a Python dictionary. In this case, the dictionary would have the Python form:

```
{
  "name":'Bob',
  "languages":["English", "French","GERMAN"]
}
```

`xsbpy` then translates this dictionary to a Prolog term, that can be pretty printed as::

```
pydict([
        ''(name,'Bob'),
        ''(languages,['English','French','GERMAN'])
        ]).
```

We call a term that maps to a Python dictionary either *a Python dictionary in term form* or just *a Prolog dictionary* although the latter slightly abuses terminology.

Note that the above example did not require writing *any* special Prolog or Python code. This is in part because Python's basic data structures – dictionaries, lists, tuples, sets and so on – are mapped to Prolog terms (cf. Section 17.3). As a result, calling Python is often a simple matter of setting up input terms for a Python function, and processing the terms that Python returns to Prolog.

Example 17.2.2 Calling a Python Function (II): Glue Code

A slightly more complex call to Python is:

```
pyfunc(xsbpy_json,prolog_load('test.json'),Ret)
```

which loads a JSON string from the file `test.json` into a Prolog term. However, the Python function `json.load()` calls requires a Python file pointer as its input, and Python file pointers do not correspond to XSB I/O streams. As we shall see in Example 17.2.4, a reference to a Python file pointer could be passed back to XSB, but in most cases it is probably easiest to write some simple Python glue code such as:

```
def prolog_load(File):
    with open(File) as fileptr:
        return(json.load(fileptr))
```

As in the previous example, the above Prolog goal produces a Prolog dictionary corresponding to the JSON file.

Example 17.2.3 Calling a Python Function (III): Keyword Arguments

Python functions often make heavy use of keyword arguments. These can be easily handled by `pyfunc/4`:

```
pyfunc(xsbpy_json,prolog_dump(Dict,'new.json'),[indent=2],Ret)
```

in which the third argument is a list of `=/2` terms. This list is turned into a Prolog dictionary and then translated into Python. Because `json.dump()` needs a file pointer in the same way `json.load()`, glue code will also needed, but the glue code passes keyword arguments in the usual manner of Python:

```
def prolog_dump(Dict,File,**Features):
    with open(File,"w") as fileptr:
        ret = json.dump(Dict,fileptr,**Features)
        return(ret)
```

The previous examples have sketched an approach that can efficiently call virtually any Python function or method, [3] although it might require a small amount of glue code. However, Python methods can also be called directly.

Example 17.2.4 Calling a Python Method

Consider the following simple Python class:

```
class Person:
  def __init__(self, name, age, ice_cream=None):
    self.name = name
    self.age = age
    if favorite_ice_cream is None:
      favorite_ice_cream = 'chocolate'
    self.favorite_ice_cream = favorite_ice_cream

  def hello(self,mytype):
    return("Hello my name is " + self.name + " and I'm a " + mytype)
```

The call

```
pyfunc('Person','Person'(john,35),Obj),
```

creates a new instance of the `Person` class, and returns a reference to this instance which has a form such as `pyObj(p0x7fb1947b0210)`. XSB can later use this reference to call a method:

```
pydot('Person',pyObj(p0x7fb1947b0210),hello(programmer),Ret2).
```

which returns the Prolog atom:

```
'Hello my name is john and I'm a programmer'
```

Although Python methods, like Python functions, can include keyword arguments, `xsbpy` does not support keyword arguments in `pydot/4` because Version 3.9.4 of the Python C API does not permit this.

Example 17.2.5 Examining a Python Object

Example 17.2.4 showed how to create a Python object, pass it back to Prolog and apply a method to it. Suppose we create another `Person` instance:

[3] `xsbpy` does not currently support Python's binary types.

```
pyfunc('Person','Person'(bob,34),Obj),
```

and later want to find out all attributes of `bob` both explicitly assigned, and default. This is easily done by `xsbpy_utils:obj_dict/2`. Assuming that `pyObj(p0x7f386e1e9650)` is the object reference for `bob` in Prolog, the call

```
obj_dict(pyObj(p0x7f386e1e9650) ,F ) .
```

returns

```
pyDict([(name,bob),(age,34),(favorite_ice_cream,chocolate)])
```

There are times when using the dictionary associated with a class is not appropriate. For instance, not all Python classes have `__dict__` methods defined for them, or only a single attribute of an object might be required. In this case, `pydot/4` can be used:

```
pydot('Person',pyObj(p0x7f386e1e9650),favorite_ice_cream,I)
```

returns `I = chocolate`.

Summarizing from Example 17.2.4 and the above paragraph, `pydot/4` can be used in two ways. If the third argument, `arg3`, in a call to `pydot/4` is a Prolog structure, `arg3` is interpreted as a method. In this case, a Python method is applied to the object, and its return is unified with the final argument of `pydot/4`. If `arg` is a Prolog atom, `arg3` is interpreted as attribute of the object. In this case, the attribute is accessed and unified with `arg3`. Note that the functionality of `pydot/4` is overloaded in analogy to the functionality of the `'.'` connector in Python.

To summarize, a great deal of Python functionality is directly available via `pyfunc/[3,4]` and `pydot/4`. In our experience so far, many Python libraries can be called directly and "just work" immediately. Cases where glue code is needed include the following.

- In a case like Example 17.2.2 where a Python method or function like `json.load()` requires a Python resource as input, a small amount of code might be useful to, say, open a file and perform an operation. However as an alternative, the file might be opened, the file pointer passed back to XSB, and the function called directly from XSB using the file pointer.
- As mentioned, `pydot/4` does not support keyword arguments, due to restrictions in the Python C API.

- Suppose a class with several attributes is defined as a subclass of, say a string type. Currently `xsbpy` will simply pass back such objects as strings, rather than as object references. An example of this occurs in the sample interface `xsbpy/apps/xsbpy_rdflib`. In the `rdflib` package `rdflib.Literal` objects are in fact subclasses of a string type. These literal objects have as attributes language tags and data types, and these attributes are critical for RDF I/O from Prolog. An example of how to handle this is seen in `xsbpy_rdflib.py`, where slightly more elaborate bridge functions are needed to marshal an object's attributes as elements of a tuple along with the object.[4]

With those disclaimers in mind, all glue code that we have needed to write so far has been simple and straightforward.

17.3 Bi-translation between Prolog Terms and Python Data Structures

`xsbpy` takes advantage of a C-level bi-translation of a large portion of Prolog terms and Python data structures: i.e., Python lists, tuples, dictionaries, sets and other objects are translated to their Prolog term forms, and Prolog terms of special syntax are translated to lists, tuples, dictionaries, sets and so on. Bi-translation is recursive in that any of these data structures can be nested in other data structures.

As mentioned above, when a Python data structure D, say a dictionary, is translated into a Prolog term T, T is sometimes called the *term form* of D. Due to a syntactic overlap between Prolog terms and Python data structures, the Prolog term forms are easy to translate and use – and sometimes appear syntactically identical.

More specifically, bi-translation between Prolog and Python can be described from the viewpoint of Python types as follows:

- *Numeric Types*: Python integers and floats are bi-translated to Prolog integers and floats. Python complex numbers are not (yet) translated, and integers are only supported if for integers between XSB's minimum and maximum integer [5]
  - *Boolean Types* which are numeric types are translated to their integer value: `True` as `1` and `False` as `0`.
- *String Types*: Python string types are bi-translated to Prolog atoms. This translation assumes UTF-8 encoding on both sides.

  Note that a Python string can be enclosed in either double quotes (`''`) or single quotes (`'`). In translating from Python to Prolog, the outer enclosure is ignored, so Python

[4] This behavior may change in future versions.

[5] These integers can be obtained by querying `current_prolog_flag/2`.

`"'Hello'"` is translated to the Prolog `'\'Hello\''`, while the Python `'"Goodby"'` is translated to the Prolog `'"Goodby"'`.

- *Sequence Types*:
  - Python lists are bi-translated as Prolog lists and the two forms are syntactically identical.
  - A Python tuple of arity `N` is bi-translated with a compound Prolog term `''/N` (i.e., the functor is the empty string, denoted by two apostrophes).
  - Python ranges are not (yet) translated (i.e., they are returned as terms with functor `pyObj/1`).
- *Mapping Types*: A Python dictionary is translated into the term form:

  `pyDict(DictList)`

  where `DictList` is a list of tuples in term form:

  `"(Key,Value)`

  `Key` and `Value` are the translations of any Python data structures that are both allowable as a dictionary key or value, and supported by `xsbpy`. For instance, `Value` can be (the term form of) a list, a set, a tuple or another dictionary.
- *Set Types*: A Python set S is translated to the term form

  `pySet(SetList)`

  where *SetList* is the list containing exactly the translated elements of S. Due to Python's implementation of sets, there is no guarantee that the order of elements will be the same in S and *SetList*.
- *None Types.* The Python keyword `None` is translated to the Prolog atom `'None'`.
- *Binary Types:* are not yet supported. There are no current plans to support this type.
- Any Python object `Obj` of a type that is not translated to a specific Prolog term as indicated above is translated to the Prolog term `pyObj(Obj)`.

Additionally, a user with a minimal knowledge of C can change parts of the syntax used in Prolog term forms. The outer functors `pyDict`, `pySet` and `pyObj` and the constant `None` can all be redefined my modifying the file `xsbpy_defs.h` in the `xsbpy` directory.

17.4 Usage

`pyfunc(+Module,+Function,+Kwargs,?Return)` module: `xsbpy`

`pyfunc(+Module,+Function,?Return)` module: `xsbpy`

Ensures that the Python module `Module` is loaded, and calls `Module.Function` unifying the return of `Function` with `Return`. A list of keyword arguments may or may not be included. For example the goal

```
pyfunc(xsbpy_rdflib,rdflib_write_file(Triples,'new_sample.ttl'),
       [format=turtle],Ret).
```

calls the function `xsbpy_rdflib,rdflib_write_file` to write `Triples`, a list of triples in Prolog format, to the file `new_sample.ttl` using the `turtle` format. This format is specified as a keyword argument to `rdflib_write_file()` in the third argument of `pyfunc/4`.

In general, `Module` must be the name of a Python module or path represented as a Prolog atom, and `Function` is the invocation of a Python function in `Module`, where `Function` is a compound Prolog structure. Optional keyword arguments are passed in the third argument as lists of `Key = Value` terms; if no such arguments are needed, `Kwargs` can be an empty list – or `pyfunc/3` may be used. Finally the return value from `Function` is unified with `Return`.

Python modules are searched for in the paths maintained in Python's `sys.path` list. As indicated below, these Python paths can be queried from XSB via `py_lib_dir/1` and modified via `add_py_lib_dir/1`.

Error Cases

- `Module` cannot be found in the current Python search paths:
  - misc_error
- `Function` does not correspond to a Python function in `Module`
  - misc_error

In addition, errors thrown by Python are caught by XSB and thrown as `misc_error` errors.

`pydot(+Module,+ObjRef,+MethAttr,?Ret)` module: `xsbpy`

Applies a method to `ObjRef`, or obtains an attribute from it. As with `pyfunc/[3,4]`, `Module` is a Python module or path. However, `ObjRef` is a Python object reference in term form (i.e., a term of the form `pyObj(Ref)` where `Ref` is a Prolog atom depicting a reference to a Python object). `pydot/4` acts in one of two ways:

- If `MethAttr` is a Prolog compound term corresponding to a Python method for `ObjRef`, the method is called and its return unified with `Ret`. Unfortunately, due to limitations in the Python C API version 3.9.4, keyword arguments cannot be used when calling Python method as they can for Python functions.

- If `MethAttr` is a Prolog atom corresponding to the name of an attribute of `ObjRef`, the attribute is accessed and unified with `Ret`.

Error Cases

- `Module` cannot be found in the current Python search paths:
  - misc_error
- `ObjRef` is not a Python object reference in Prolog term form:
  - misc_error
- `MethAttr` is neither a Prolog compound term nor a Prolog atom:
  - misc_error

In addition, errors thown by Python are caught by XSB and re-thrown as `misc_error` errors.

`pp_py(+Stream,+Term)` module: `xsbpy`
`pp_py(Term)` module: `xsbpy`

Pretty prints the Prolog translation of a Python data structure in Python-like syntax. For instance, the term

```
pydict([''(name,'Bob'),''(languages,['English','French','GERMAN'])]).
```

is printed as

```
{
  name:'Bob',
  languages:[
   'English','
   'French',
   'GERMAN'
  ]
}
```

Such pretty printing can be useful for developing applications such as the `xsbpy` Elasticsearch interface.

`add_py_lib_dir(+Path)` module: `xsbpy`

This convenience predicate allows the user to add a path to the Python library directories in a manner similar to `add_lib_dir/1`, which adds Prolog library directories.

`py_lib_dirs(?Path)` module: `xsbpy`

This convenience predicate returns the current Python library directories as a Prolog list.

`values(+Dict,+Path,?Val)` module: `xsbpy_utils`

Convenience predicate to obtain a value from a (possibly nested) Prolog dictionary. The goal

`values(D,key1,V)`

is equivalent to the Python expression `D[key1]` while

`values(D,[key1,key2,key3],V)`

is equivalent to the Python expression

`D[key1][key2][key3]`.

There are no error conditions associated with this predicate.

`keys(+Dict,?Keys)` module: `xsbpy_utils`
`items(+Dict,?Items)` module: `xsbpy_utils`

Convenience predicates (for the inveterate Python programmer) to obtain a list of keys or items from a Prolog dictionary. There are no error conditions associated with these predicates.

`obj_dict(+ObjRef,-Dict)` module: `xsbpy_utils`

Given a reference to a Python object as `ObjRef`, this predicate returns the dictionary of attributes of `ObjRef` in `Dict`. If no ___dict___ attribute is associated with `ObjRef` the predicate fails.

`obj_dict/2` is a convenience predicate, and could be written using `pydot/4` as:

```
pydot('__main__',Obj,'__dict__',Dict).
```

`obj_dir(+ObjRef,-Dir)` module: `xsbpy_utils`

Given a reference to a Python object as `ObjRef`, this predicate returns the list of attributes of `ObjRef` in `Dir`. If no ___dir___ attribute is associated with `ObjRef` the predicate fails.

`obj_dir/2` is a convenience predicate, and could be written using `pydot/4` as:

```
pydot('__main__',Obj,'__dir__'(),Dir).
```

17.5 Allowing Python to Reclaim Space

TBD: Discuss space issues for Python how they are now addressed and how they will be addressed in the future.

17.6 Performance

The core `xsbpy` routines – `pyfunc/[3,4]` and `pydot/4` – are written almost entirely in C, have shown good performance so far, and continue to be optimized. Calling a simple Python function to increment a number from XSB and then returning the incremented value to XSB should take about a microsecond on a reasonably fast machine. Of course, the overhead for passing large terms from and to Python will be somewhat higher. For instance, the time to pass a list of integers from Python to XSB has been timed at about 20-30 nanoseconds per list element. Nonetheless, for nearly any practical application the time to perform useful functionality within Python will far outweigh any `xsbpy` overhead.

Apart from system resource limitations, there is virtually no upper limit on the size of Python structures passed back to Prolog: stress tests have passed lists of length 100 million from Python to Prolog without problems. However, it should be noted that `xsbpy` must ensure that XSB's heap is properly expanded before copying a large structure from Python to XSB. XSB currently uses size routines from the Python C-API which only return the length of structures, and not their exact size. Accordingly long lists of large structures, heavily nested dictionaries and other such structures may currently present a problem. This will be addressed in a future version, but until then a user can reset the `heap_margin` Prolog flag to ensure extra stack space beyond that allocated by `xsbpy` (see Volume 1 for details.)

17.7 Sample Applications

When `xsbpy` is loaded, both the `xsbpy` directory and its subdirectories, `apps` and `examples`, are added to the Prolog and Python paths. As a result, modules in these subdirectories can be loaded into XSB and Python without changing their library paths.

Some of these modules, like `xsbpy_json` and `xsbpy_rdflib` provide useful functionality and can be used as written, or nearly so; other modules are included to help the user get started in writing hybrid applications.

Note that several of these applications require the underlying Python libraries to have been installed via a `pip` or `conda` install.

(Rudimentary) Applications

xsbpy_json This module contains an interface to the Python `json` module, with predicates to read JSON from and write JSON to files and strings. The `json` module transforms JSON objects into and from Python dictionaries, which the interface maps to and from their term forms.

xsbpy_rdflib This module interfaces to the Python `rdflib` library to read RDF information from files in Turtle, N-triples and N-quads format, and to write files in Turtle and N-triples format. As such it augments XSB's RDF package (Chapter 10) which handles XML-RDF. [6]

xp_spacy The Prolog and Python files for `xp_spacy` have predicates for loading a Spacy model into Python and using that model to make a Spacy document structure. Other predicates load much of the information from Spacy into a Prolog graph, and show how to navigate the Prolog graph to reconstruct sentences, illustrate the parsed dependency graph, and so on.

xp_faiss The dense-vector query engine Faiss [7], developed by Facebook offers an efficient way to perform nearest neighbor searches in vector spaces produced by word, network, tuple, or other embeddings. The `xp_faiss` example provides XSB predicates to initialize a Faiss index from a text file of vectors, perform queries to the index, and to make a weighted Prolog graph out of the vector space.

As with many machine-learning tooks, Faiss expects that each of the vectors is referenced by an integer. For instance, a vector for the string *cheugy* would be referenced by an integer, say 37. The XSB programmer thus would be responsible for associating the string *cheugy* with 37 in order to use Faiss. The main predicates exported by `xp_faiss.P` include:

- `faissInit(+XbFile,+Dim)` initizlizes a Faiss index where `XBFile` is a text file containing the vectors to be indexed and `Dim` is the dimension of these vectors. (`xb` is Faiss terminology for the set of indexed vectors.) This predicate also creates a `numpy` array with a set of query vectors `xq` consisting of the same vectors. With the query and index vectors both set up in this manner, the vector space can be explored to understand which of its elements are suitable for clustering, and so on.

  After execution of this predicate, a fact for the predicate `xp_faiss:xq_num/1` contains the number of query vectors (xq), which is the same as the number of indexed vectors (xb).

- `get_k_nn(+Node,+K,-Neighbors)` finds the K nearest neighbors of a node. The predicate takes as input `Node`, the integer identifier of a node, and `K` the number of nearest neighbors to be returned. The return structure `Neighbors` is the Prolog representation of a 2-ary Python tuple (i.e., `"/2`) containing as its first argument a list of `K` distances and as it second argument a list of `K` neighbors.

[6] Testing has been done of this `xsbpy` interface, but the testing has not been exhaustive. As a result, please double-check its results if it is used, and report bugs to `xsb.sorceforge.net`.

- `make_vector_graph(K)` Given a Faiss index, this predicate asserts a weighted graph in Prolog by obtaining the nearest `K` neighbors for each indexed vector. Edges of the graph have the form:

  `vector_edge_raw(From,To,Dist)`

  Where `From` and `To` are integer referents for indexed vectors, and `Dist` is the Euclidean distance between the vector with referent `From` and the vector with referent `To`. Each fact of `vector_edge_raw/3` is indexed both on its first and second argument.

  If both the number of indexed vectors and `K` are large, construction of the Prolog vector graph may take a few minutes. Construction time is almost wholly comprised of the time to find the set of nearest neighbors for each node.

- `vector_edge(Node1,Node2,Dist)`. The vector graph, which represents distances is undirected. However to save space, the `vector_edge_raw/3` facts are asserted so that if `vector_edge_raw(Node1,Node2,Dist)` has been asserted, `vector_edge_raw(Node2,Node1,Dist)` will not be asserted. `vector_edge/3` calls `vector_edge_raw/3` in both directions, and should be used for querying the vector graph.

- `write_vector_graph(+File,+Header)` writes out the vector graph to `File`. This predicate ensures that `File` contains the proper indexing directive for `vector_edge_raw/3` as well a directive to the compiler describing how to dynamically load `File` in an efficient manner. Because of these directives, the file can simply be consulted or ensure_loaded and the user does not need to worry about which compiler options should be used. The graph is loaded into the module `vector_graph`.

  `Header` is simply a string that is written as a comment to the first line of `File` that can serve to contain any necessary provenance information.

Other Examples

xsbpy_elastic This module contains sample code for using the Python `elasticsearch` module. A step by step description shows how a connection is opened, and index is created and a document added and committed. The example then shows how the document can be searched in two ways, and finally deleted.

Much of the information that Elasticsearch reads and writes is in JSON format, which the Python interface transforms to dictionaries, and `xsbpy` transforms these dictionaries to and from their term form. Thus although this example is short, the ideas in it can easily be extended to a full interface. [7]

Fasttext Language Detection: xp_fasttext This example provides some code to help the user get started using Fasttext's language detection, and relies on the python mod-

[7] This has already been done by one company that uses XSB.

ule `fasttext`. Assuming that Fasttext's language identification module is in the current directory, the command:

```
pyfunc(fasttext,load_model('./lid.176.bin'),Obj).
```

Loads the model and unifies `Obj` with a reference to the loaded module which might look like `pyObj(p0x7faca3428510)`. Note that this can be done by calling the Python `fasttext` module directly. Next, a call to the example python module `xp_fasttext`:

```
pyfunc(xp_fasttext, fasttext_predict(pyObj(p0x7faca3428510),
       'xsbpy is a really useful addition to XSB! But language detection
        requires a longer string than I usually want to type.'),Lang).
```

returns the detected language and confidence value, which in this case is `(__label__en,0.93856)`. The only reason that the module `xp_fasttext` needs to be used is because the confidence is returned as a `numpy` array, which `xsbpy` does not currently translate automatically. [8]

googleTrans This example provides code to access Google's web-services for language translation and language detection.

17.8 Current and Future Work

- A callback mechanism is under development. This mechanism allows XSB and Python to recursively call each other. Our intention is to make our callback mechanism consistent with PyXSB, `pypi.org/project/py-xsb`, but currently PyXSB and `xsbpy` are independent of each other.
- A possible future version may include a hook in XSB's atom garbage collection to list Python objects that may be garbage collected, and to send this information back to Python.

[8] The examples `xp_fasttext` and `googleTrans` were written by Albert Ki; `xp_faiss` was written by ALbert Ki and Theresa Swift.

Chapter 18

XASP: Answer Set Programming with XSB and Smodels

By Luis Castro, Theresa Swift, David S. Warren [1]

The term *Answer Set Programming (ASP)* describes a paradigm in which logic programs are interpreted using the (extended) stable model semantics. While the stable model semantics is quite elegant, it has radical differences from traditional program semantics based on Prolog. First, stable model semantics applies only to ground programs; second stable model semantics is not goal-oriented – determining whether a stable model is true in a program involves examining each clause in a program, regardless of whether the goal would depends on the clause in a traditional evaluation.

Despite (or perhaps because of) these differences, ASP has proven to be a useful paradigm for solving a variety of combinatorial programs. Indeed, determining a stable model for a logic program can be seen as an extension of the NP-complete problem of propositional satisfiability, so that satisfiability problems that can be naturally represented as logic programs can be solved using ASP.

The current generation of ASP systems are very efficient for determining whether a program has a stable model (analogous to whether the program, taken as a set of propositional axioms, is satisfiable). However, ASP systems have somewhat primitive file-based interfaces. XSB is a natural complement to ASP systems. Its basis in Prolog provides a procedural counterpart for ASP, as described in Chapter 5 of Volume 1 of this manual; and XSB's computation of the Well-founded semantics has a well-defined relationship to stable model semantics. Furthermore, deductive-database-like capabilities of XSB allow it to be an efficient and flexible grounder for many ASP problems.

The XASP package provides various mechanisms that allow tight linkage of XSB pro-

[1] Thanks to Barry Evans for helping resuscitate the XASP installation procedure, and to Gonçalo Lopes for the installation procedure on Windows.

grams to the Smodels [9] stable model generator. The main interface is based on a store of clauses that can be incrementally asserted or deleted by an XSB program. Clauses in this store can make use of all of the cardinality and weight constraint syntax supported by Smodels, in addition to default negation. When the user decides that the clauses in a store are a complete representation of a program whose stable model should be generated, the clauses are copied into Smodels buffers. Using the Smodels API, the generator is invoked, and information about any stable models generated are returned. This use of XASP is roughly analogous to building up a constraint store in CLP, and periodically evaluating that store, but integration with the store is less transparent in XASP than in CLP. In XASP, clauses must be explicitly added to a store and evaluated; furthermore clauses are not removed from the store upon backtracking, unlike constraints in CLP.

The XNMR interpreter provides a second, somewhat more implicit use of XASP. In the XNMR interface a query Q is evaluated as is any other query in XSB. However, conditional answers produced for Q and for its subgoals, upon user request, can be considered as clauses and sent to Smodels for evaluation. In backtracking through answers for Q, the user backtracks not only through answer substitutions for variables of Q, but also through the stable models produced for the various bindings.

18.1 Installing the Interface

Installing the Smodels interface of XASP sometimes can be tricky for two reasons. First, XSB must dynamically load the Smodels library, and dynamic loading introduces platform dependencies. Second since Smodels is written in C++ and XSB is written in C, the load must ensure that names are properly resolved and that C++ libraries are loaded, steps that may addressed differently by different compilers [2]. However, by following the steps outlined below in the section for Unix or Windows, XASP should be running in a matter of minutes.

18.1.1 Installing the Interface under Unix

In order to use the Smodels interface, several steps must be performed.

1. *Creating a library for Smodels.* Smodels itself must be compiled as a library. Unlike previous versions of XSB, which required a special configuration step for Smodels, Version 4.0 requires no special confiuration, since XSB includes source code for Smodels 2.33 as a subdirectory of the `$XSBDIR/packages/xasp` directory (denoted `$XASPDIR`).

[2] XSB's compiler can automatically call foreign compilers to compile modules written in C, but in Version 4.0 of XSB C++ modules must be compiled with external commands, such as the `make` command shown below.

We suggest making Smodels out of this directory [3]. Thus, to make the Smodels library

(a) Change directory to `$XASPDIR/smodels`

(b) On systems other than OS X, type

```
make lib
```

on OS X, type [4]

```
make -f Makefile.osx lib
```

If the compilation step ran successfully, there should be a file `libsmodels.so` (or `libsomodels.dylib` on MacOS X or `libsmodels.dll` on Windows...) in `$XASPDIR/smodels/.libs`

(c) Change directory back to `$XASPDIR`

2. *Compiling the XASP files* Next, platform-specific compilation of XASP files needs to be performed. This can be done by consulting `prologMake.P` and executing the goal

```
?- make.
```

3. *Checking the Installation* To see if the installation is working properly, cd to the subdirectory `tests` and type:

```
sh testsuite.sh <$XSBDIR>
```

If the test suite succeeded it will print out a message along the lines of

```
PASSED testsuite for /Users/tswift/XSBNEW/XSB/config/powerpc-apple-darwin7.5.1/bin/xsb
```

18.1.2 Installing XASP under Windows using Cygwin

To install XASP under Windows, you must use Version 4.0 of XSB or later and Version 2.31 or later of Smodels [5]. You should also have a recent version of Cygwin (e.g. 1.5.20 or later) with all the relevant development packages installed, such as `devel`, `make`, `automake`, `patchtools`, and possibly `x11` (for `makedepend`) Without an appropriate Cygwin build environment many of these steps will simply fail, sometimes with quite cryptic error messages.

[3] Although distributed with XSB, Smodels is distributed under the GNU General Public License, a license that is slightly stricter than the license XSB uses. Users distributing applications based on XASP should be aware of any restrictions imposed by GNU General Public License.

[4] A special makefile is needed for OS X since the GNU libtool is called glibtool on this platform.

[5] This section was written by Goncalo Lopes.

1. *Patch and Compile Smodels* First, uncompress `smodels-2.31.tar.gz` in some directory, (for presentation purposes we use `/cygdrive/c/smodels-2.31` — that is, `c:\smodels-2.31`). After that, you must apply the patch provided with this package. This patch enables the creation of a DLL from Smodels. Below is a sample session (system output omitted) with the required commands:

   ```
   $ cd /cygdrive/c/smodels-2.31
   $ cat $XSB/packages/xasp/patch-smodels-2.31 | patch -p1
   $ make lib
   ```

   After that, you should have a file called `smodels.dll` in the current directory, as well as a file called `smodels.a`. You should make the former "visible" to Windows. Two alternatives are either (a) change the `PATH` environment variable to contain `c:\smodels-2.31`, or (b) copy `smodels.dll` to some other directory in your PATH (such as `c:\windows`, for instance). One simple way to do this is to copy `smodels.dll` to `$XSB/config/i686-pc-cygwin/bi` *after* the configure XSB step (step 2), since that directory has to be in your path in order to make XSB fully functional.

2. *Configure XSB.* In order to properly configure XSB, you must tell it where the Smodels sources and library (the `smodels.a` file) are. In addition, you must compile XSB such that it doesn't use the Cygwin DLL (using the `-mno-cygwin` option for gcc). The following is a sample command:

   ```
   $ cd $XSB/build
   $ ./configure --enable-no-cygwin -with-smodels="/cygdrive/c/smodels-2.31''
   ```

   You can optionally include the extended Cygwin w32 API using the configuration option `--with-includes=<PATH_TO_API>`, (this allows XSB's build procedure to find `makedepend` for instance), but you'll probably do fine with just the standard Cygwin apps.

   There are some compiler variables which may not be automatically set by the configure script in `xsb_config.h`, namely the configuration names and some activation flags. To correct this, do the following:

   (a) cd to `$XSB/config/i686-pc-cygwin`

   (b) open the file `xsb_config.h` and add the following lines:

   ```
   #define CONFIGURATION "i686-pc-cygwin"
   #define FULL_CONFIG_NAME "i686-pc-cygwin"
   #define SLG_GC
   ```

   (Still more flags may be needed depending on Cygwin configuration)

   After applying these changes, cd back to the `$XSB/build` directory and compile XSB:

```
$ ./makexsb
```

Now you should have in `$XSB/config/i686-pc-cygwin/bin` directory both a `xsb.exe` and a `xsb.dll`.

3. *Compiling XASP.* First, go to the XASP directory and execute the `makelinks.sh` script in order to make the headers and libraries in Smodels be accessible to XSB, i.e.:

```
$ cd $XSB/packages/xasp
$ sh makelinks.sh /cygdrive/c/smodels-2.31
```

Now you must copy the `smoMakefile` from the `config` directory to the `xasp` directory and run both its directives:

```
$ cp $XSB/config/i686-pc-cygwin/smoMakefile .
$ make -f smoMakefile module
$ make -f smoMakefile all
```

At this point, you can consult `xnmr` as you can with any other package, or xsb with the `xnmr` command line parameter, like this: (don't forget to add XSB bin directory to the `$PATH` environment variable)

```
$ xsb xnmr
```

Lots of error messages will probably appear because of some runtime load compiler, but if everything goes well you can ignore all of them since your xasppkg will be correctly loaded and everything will be functioning smoothly from there on out.

18.2 The Smodels Interface

The Smodels interface contains two levels: the *cooked* level and the *raw* level. The cooked level interns rules in an XSB *clause store*, and translates general weight constraint rules [17] into a *normal form* that the Smodels engine can evaluate. When the programmer has determined that enough clauses have been added to the store to form a semantically complete sub-program, the program is *committed*. This means that information in the clauses is copied to Smodels and interned using Smodels data structures so that stable models of the clauses can be computed and examined. By convention, the cooked interface ensures that the atom `true` is present in all stable models, and the atom `false` is false in all stable models. The raw level models closely the Smodels API, and demands, among other things, that each atom in a stable sub-program has been translated into a unique integer. The raw level also does not provide translation of arbitrary weight constraint rules into the normal form required by

the Smodels engine. As a result, the raw level is significantly more difficult to directly use than the cooked level. While we make public the APIs for both the raw and cooked level, we provide support only for users of the cooked interface.

As mentioned above Smodels extends normal programs to allow weight constraints, which can be useful for combinatorial problems. However, the syntax used by Smodels for weight constraints does not follow ISO Prolog syntax so that the XSB syntax for weight constraints differs in some respects from that of Smodels. Our syntax is defined as follows, where *A* is a Prolog atom, *N* a non-negative integer, and *I* an arbitrary integer.

- *GeneralLiteral ::= WeightConstraint | Literal*
- *WeightConstraint ::= weightConst(Bound,WeightList,Bound)*
- *WeightList ::= List of WeightLiterals*
- *WeightLiteral ::= Literal | weight(Literal,N)*
- *Literal ::= A | not(A)*
- *Bound ::== I |* `undef`

Thus an example of a weight constraint might be:

- `weightConst(1,[weight(a,1),weight(not(b),1)],2)`

We note that if a user does not wish to put an upper or lower bound on a weight constraint, she may simply set the bound to `undef` or to an integer less than `0`.

The intuitive semantics of a weight constraint `weightConst(Lower,WeightList,Upper)`, in which `List` is is list of *WeightLiterals* that it is true in a model *M* whenever the sum of the weights of the literals in the constraint that are true in *M* is between the lower `Lower` and `Upper`. Any literal in a *WeightList* that does not have a weight explicitly attached to it is taken to have a weight of *1*.

In a typical session, a user will initialize the Smodels interface, add rules to the clause store until it contains a semantically meaningful sub-problem. He can then specify a compute statement if needed, commit the rules, and compute and examine stable models via backtracking. If desired, the user can then re-initialize the interface, and add rules to or retract rules from the clause store until another semantically meaningful sub-program is defined; and then commit, compute and examine another stable model [6].

The process of adding information to a store and periodically evaluating it is vaguely reminiscent of the Constraint Logic Programming (CLP) paradigm, but there are important differences. In CLP, constraints are part of the object language of a Prolog program:

[6] Currently, only normal rules can be retracted.

constraints are added to or projected out of a constraint store upon forward execution, removed upon backwards execution, and iteratively checked. When using this interface, on the other hand, an XSB program essentially acts as a compiler for the clause store, which is treated as a target language. Clauses must be explicitly added or removed from the store, and stable model computation cannot occur incrementally – it must wait until all clauses have been added to the store. We note in passing that the `xnmr` module provides an elegant but specialized alternative. `xnmr` integrates stable models into the object language of XSB, by computing ""relevant"" stable models from the the residual answers produced by query evaluation. It does not however, support the weighted constraint rules, compute statements and so on that this module supports.

Neither the raw nor the cooked interface currently supports explicit negation.

Examples of use of the various interfaces can be found in the subdirectory `intf_examples`

`smcInit`
: Initializes the XSB clause store and the Smodels API. This predicate must be executed before building up a clause store for the first time. The corresponding raw predicate, `smrInit(Num)`, initializes the Smodels API assuming that it will require at most `Num` atoms.

`smcReInit`
: Reinitializes the Smodels API, but does *not* affect the XSB clause store. This predicate is provided so that a user can reuse rules in a clause store in the context of more than one sub-program.

`smcAddRule(+Head,+Body)`
: Interns a ground rule into the XSB clause store. `Head` must be a *GeneralLiteral* as defined at the beginning of this section, and `Body` must be a list of *GeneralLiterals*. Upon interning, the rule is translated into a normal form, if necessary, and atoms are translated to unique integers. The corresponding raw predicates, `smrAddBasicRule/3`, `smrAddChoiceRule/3`, `smrAddConstraintRule/4`, and `smrAddWeightRule/3` can be used to add raw predicates immediately into the SModels API.

`smcRetractRule(+Head,+Body)`
: Retracts a ground (basic) rule from the XSB clause store. Currently, this predicate cannot retract rules with weight constraints: `Head` must be a *Literal* as defined at the beginning of this section, and `Body` must be a list of *GeneralLiterals*.

`smcSetCompute(+List)`
: Requires that `List` be a list of literals – i.e. atoms or the default negation of atoms). This predicate ensures that each literal in `List` is present in the stable models returned by Smodels. By convention the cooked interface ensures that `true` is present and `false` absent in all stable models. After translating a literal it calls the raw interface predicates `smrSetPosCompute/1` and `smrSetNegCompute/1`

`smcCommitProgram`
: This predicate translates all of the clauses from the XSB clause store into the data structures of the Smodels API. It then signals to the API that all clauses have been added, and initializes the Smodels computation. The corresponding raw predicate, `smrCommitProgram`, performs only the last two of these features.

`smComputeModel`
: This predicate calls Smodels to compute a stable model, and succeeds if a stable model can be computed. Upon backtracking, the predicate will continue to succeed until all stable models for a given program cache have been computed. `smComputeModel/0` is used by both the raw and the cooked levels.

`smcExamineModel(+List,-Atoms)`
: `smcExamineModel/(+List,-Atoms)` filters the literals in `List` to determine which are true in the most recently computed stable model. These true literals are returned in the list `Atoms`. `smrExamineModel(+N,-Atoms)` provides the corresponding raw interface in which integers from `0` to `N`, true in the most recently computed stable model, are input and output.

`smEnd`
: Reclaims all resources consumed by Smodels and the various APIs. This predicate is used by both the cooked and the raw interfaces.

`print_cache`
: This predicate can be used to examine the XSB clause store, and may be useful for debugging.

18.3 The xnmr_int Interface

. This module provides the interface from the `xnmr` module to Smodels. It does not use the `sm_int` interface, but rather directly calls the Smodels C interface, and can be thought of as a special-purpose alternative to `sm_int`.

`init_smodels(+Query)`
: Initializes smodels with the residual program produced by evaluating `Query`. `Query` must be a call to a tabled predicate that is currently completely evaluated (and should have a delay list)

`atom_handle(?Atom,?AtomHandle)`
: The *handle* of an atom is set by `init_smodels/1` to be an integer uniquely identifying each atoms in the residual program (and thus each atom in the Herbrand base of the program for which the stable models are to be derived). The initial query given to `init_smodels` has the atom-handle of 1.

`in_all_stable_models(+AtomHandle,+Neg)`
: `in_all_stable_models/2` returns true if `Neg` is 0 and the atom numbered `AtomHandle` returns true in all stable models (of the residual program set by the previous call to `init_smodels/1`). If `Neg` is nonzero, then it is true if the atom is in NO stable model.

`pstable_model(+Query,-Model,+Flag)`
: returns nondeterministically a list of atoms true in the partial stable model total on the atoms relevant to instances of `Query`, if `Flag` is 0. If `Flag` is 1, it only returns models in which the instance of `Query` is true.

`a_stable_model`
: This predicate invokes Smodels to find a (new) stable model (of the program set by the previous invocation of `init_smodels/1`.) It will compute all stable models through backtracking. If there are no (more) stable models, it fails. Atoms true in a stable model can be examined by `in_current_stable_model/1`.

`in_current_stable_model(?AtomHandle)`
: This predicate is true of handles of atoms true in the current stable model (set by an invocation of `a_stable_model/0`.)

`current_stable_model(-AtomList)`
: returns the list of atoms true in the current stable model.

`print_current_stable_model`
: prints the current stable model to the stream to which answers are sent (i.e `stdfbk`)

Chapter 19

Importing and Exporting JSON Structures

by Michael Kifer

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

19.1 Introduction

In brief, a JSON structure is a *value*, which can be 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 the 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 `//`. The JSON parser 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 JSON parser does not allow duplicate keys and considers such objects as ill-formed. However, it also provides an option to allow duplicate keys.

19.2 API for Importing JSON as Terms

When XSB ingests a JSON structure, it represents it as a term as follows:

- Arrays are represented as lists.
- Strings are represented as 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'(_)`.
- Finally, an object of the form `{` $str_1 : val_1, \ldots, str_n : val_n$ `}` is represented as `json([`str'_1=$val'_1, \ldots, str'_n$ where str'_i is the atom corresponding to the string str_i and val'_i is the XSB representation of the JSON value val_i. Here `json` is a unary Prolog function symbol.

For instance, the above examples would be represented as Prolog terms as follows:

```
json([first = John, last = Doe, age = 25])
[1, 2, json([one = 1.1000, two = 2.2200]), 'NULL'(\_)]
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).

XSB provides the following methods for importing JSON:

- `parse_json(`*`Source,Result`*`)`
  Here *Source* can have one of these forms
  - `string(`*`Atom`*`)`
  - `atom(`*`Atom`*`)`
  - `url(`*`Atom`*`)`
  - `file(`*`Atom`*`)`
  - *Atom*
  - a variable

  The forms `string(`*`Atom`*`)` and `atom(`*`Atom`*`)` must supply an atom whose content is a JSON structure and *Result* will then be bound to the XSB representation of that structure. The form `url(`*`Atom`*`)` can be used to ask XSB 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 when the entire term has been entered. 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 a Prolog term as explained above. For instance, the query `parse_json(string('{"abc":1, "cde":2}'),X)` will bind `X` to the XSB term `json([abc=1,cde=2])` while the query `parse_json(string('{"abc":1, "cde":2}'),pretty(X))` will bind `X` to the atom

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

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

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

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

  then the query `parse_json(_,first,X)` will bind `X` to 1 while `parse_json(_,'second.fourth',X)` will bind it to `json([fifth = 3])`.

  Note that the selector lets one navigate through subobjects but 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_json(_,'second.fourth',X)` 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`*`)`
  This sets options for parsing JSON for all the subsequent calls to the JSON parser. Currently, only the following options are 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 John Doe. To make the output a bit more manageable, we are only asking to get the JSON subobject rooted at the property `itemListElement`. (The Google KG's session key in the example is invalid: one must supply one's own key.)

```
?- U =
'https://kgsearch.googleapis.com/v1/entities:search?query=john_doe&key=XYZ&limit=1',
   parse_json(url(U), itemListElement, Answer).
```

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

19.3 Exporting Terms to JSON

An exported term is represented simply as a JSON object with two features: *functor* and *arguments*. The *arguments* part is a list of terms and these terms are converted to JSON recursively, by the same rule. For instance,

```
| ?- term_to_json(ppp(a(9),b,L,[pp(ii),2,3,L],K),J).
J = '{"functor":"ppp","module":"usermod",
      "arguments":[{"functor":"a","module":"usermod","arguments":[9]},
                   "b",
                   {"variable":"_h0"},
                   [{"functor":"pp","module":"usermod","arguments":["ii"]},
                    2,
                    3,
                    {"variable":"_h0"}],
                   {"variable":"_h1"}]}'

| ?- term_to_json(foo(a,b,bar(c,d)),J).
J = '{"functor":"foo","module":"usermod",
      "arguments":["a","b",
                   {"functor":"bar","module":"usermod",
                    "arguments":["c","d"]}]}'

| ?- term_to_json((a,b,bar(c,d)),J).
J = '{"commalist":["a","b",
      {"functor":"bar","module":"usermod","arguments":["c","d"]}]}'
```

Backslashes and double quotes that are part of exported strings are escaped with additional backslashes, as required by JSON. For instance

```
| ?- term_to_json('foo\goo"moo'('bar\ggg123"456'),J).
J = '{"functor":"foo\\goo\"moo","module":"usermod","arguments":["bar\\ggg123\"456"]}'
```

Bibliography

[1] C. Draxler. Prolog to SQL compiler, Version 1.0. Technical report, CIS Centre for Information and Speech Processing Ludwig-Maximilians-University, Munich, 1992.

[2] D. Dubois, J. Lang, and H. Prade. Towards possibilistic logic programming. In *ICLP*, pages 581–595, 1991.

[3] D. Dubois, J. Lang, and H. Prade. Possibilistic logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of logic in artificial intelligence and logic programming,vol. 3*, pages 439–514. Oxford University Press, 1994.

[4] T. Fruehwirth. Thom Fruehwirth's Constraint Handling Rules website. http://www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/chr-intro.html.

[5] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.

[6] C. Holzbaur. Ofai clp(q,r) manual, edition 1.3.3. Technical report, Austrian Research Institute for Artificial Intelligence, 1995.

[7] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[8] T. I. S. Laboratory. *SICStus Prolog User's Manual Version 3.12.5*. Swedish Institute of Computer Science, 2006.

[9] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.

[10] F. Riguzzi and T. Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming*, 11(4-5):433–449, 2011.

[11] F. Riguzzi and T. Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming*, 13(2):279–302, 2013.

[12] F. Riguzzi and T. Swift. A survey of probabilistic logic programming. In *Declarative Logic Programming: Theory, Systems, and Applications*, volume 20 of *ACM Books*, pages 185–233, 2018.

[13] B. Sanna-Starosta. Chrd: A set-based solver for constraint hanlding rules. available at www.cs.msu.edu/˜bss/chr-d, 2006.

[14] B. Sanna-Starosta and C. Ramakrishnan. Compiling constraint handling rules for efficient tabled evaluation. available at www.cs.msu.edu/˜bss/chr-d, 2006.

[15] T. Sato. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*, pages 715–729. MIT Press, 1995.

[16] T. Sato and Y. Kameya. Prism: A language for symbolic-statistical modeling. In *International Joint Conference on Artificial Intelligence*, pages 1330–1339, 1997.

[17] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[18] J. Vennekens, M. Denecker, and M. Bruynooghe. Representing causal information about a probabilistic process. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence*, LNAI. Springer, September 2006.

[19] J. Vennekens, M. Denecker, and M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory Pract. Log. Program.*, 9(3):245–308, 2009.

[20] J. Vennekens and S. Verbaeten. Logic programs with annotated disjunctions. Technical Report CW386, K. U. Leuven, 2003.

[21] J. Vennekens, S. Verbaeten, and M. Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming*, volume 3131 of *LNCS*, pages 195–209. Springer, 2004.

[22] J. Wielemaker. *SWI Prolog version 5.6: Reference Manual*. University of Amsterdam, 2007.

Index