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'

2.2.11 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).

2.2.12 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 on 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,Backtra 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.

2.2.13 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.

2.3 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 in the 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

2.4 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.

8.4 XPath support 87

9 rdf : The XSB RDF Parser 90
9.1 Introduction 90
9.2 High-level API 90
9.2.1 RDF Object representation 91
9.2.2 Name spaces 92
9.2.3 Low-level access 92
9.3 Testing the RDF translator 93

10 Constraint Packages 94
10.1 clpr : The CPL(R) package 94
10.1.1 The CLP(R) API 95
10.2 The bounds Package 100
10.2.1 The bounds API 102

11 Constraint Handling Rules 105
11.1 Introduction 105
11.2 Syntax and Semantics 105
11.2.1 Syntax 105
11.2.2 Semantics 106
11.3 CHR in XSB Programs 108
11.3.1 Embedding in XSB Programs 108
11.3.2 Compilation 108
11.4 Useful Predicates 109
11.5 Examples 109
11.6 CHR and Tabling 110
11.6.1 General Issues and Principles 110
11.6.2 Call Abstraction 111
11.6.3 Answer Projection 111
11.6.4 Answer Combination 113
11.6.5 Overview of Tabling-related Predicates 115
11.7 Guidelines 115

Then we spawn another process, sort, which picks the output from the first process (since it uses the stream FromCat1 as its input) and sends its own output (the sorted version of data) to its output stream Stream. However, Stream has already been open for output into the file test. Thus, the overall result of the above clause is tantamount to the following shell command:

```
cat data | sort > test
```

Important notes about spawned processes:

1. Asynchronous processes spawned by XSB do not disappear (at least on Unix) when they terminate, unless the XSB program executes a wait on them (see process_control below). Instead, such processes become defunct zombies (in Unix terminology); they do not do anything, but consume resources (such as file descriptors). So, when a subprocess is known to terminate, it must be waited on.

2. The XSB parent process must know how to terminate the asynchronous subprocesses it spawns. The drastic way is to kill it (see process_control below). Sometimes a subprocess might terminate by itself (e.g., having finished reading a file). In other cases, the parent and the child programs must agree on a protocol by which the parent can tell the child to exit. The programs in the XSB subdirectory examples/subprocess illustrate this idea. If the child subprocess is another XSB process, then it can be terminated by sending the atom end_of_file or halt to the standard input of the child. (For this to work, the child XSB must waiting at the prompt).

3. It is very important to not forget to close the streams that the parent uses to communicate with the child. These are the streams that are provided in arguments 2,3,4 of spawn_process. The reason is that the child might terminate, but these streams to the standard input of the child will remain open, since they belong to the parent process. As a result, the parent will own defunct I/O streams and might eventually run out of file descriptors or streams.

process_status(+Pid,-Status)

This predicate always succeeds. Given a process id, it binds the second argument (which must be an unbound variable) to one of the following atoms: running, stopped, exited_normally, exited_abnormally, aborted, invalid, and unknown. The invalid status is given to processes that never existed or that are not children of the parent XSB process. The unknown status is assigned when none of the other statuses can be assigned.

Note: process status (other than running) is system dependent. Windows does not seem to support stopped and aborted. Also, processes killed using the process_control predicate (described next) are often marked as invalid rather than exited, because Windows seems to lose all information about such processes. Process status might be inaccurate in some Unix systems as well, if the process has terminated and wait() has been executed on that process.

process_control(+Pid,+Operation)

Perform a process control operation on the process with the given Pid. Currently, the only supported operations are kill (an atom) and wait(Code) (a term). The former causes the

process to exit unconditionally, and the latter waits for process completion. When the process exits, Code is bound to the process exit code. The code for normal termination is 0.

This predicate succeeds, if the operation was performed successfully. Otherwise, it fails. The wait operation fails if the process specified in Pid does not exist or is not a child of the parent XSB process.

The kill operation might fail, if the process to be killed does not exist or if the parent XSB process does not have the permission to terminate that process. Unix and Windows have different ideas as to what these permissions are. See kill(2) for Unix and TerminateProcess for Windows.

Note: under Windows, the programmer's manual warns of dire consequences if one kills a process that has DLLs attached to it.

get_process_table(-ProcessList) module: shell

This predicate is imported from module shell. It binds ProcessList to the list of terms, each describing one of the active XSB subprocesses (created via spawn_process/5). Each term has the form:

process(Pid,ToStream,FromStream,StderrStream,CommandLine).

The first argument in the term is the process id of the corresponding process, the next three arguments describe the three standard streams of the process, and the last is an atom that shows the command line used to invoke the process. This predicate always succeeds.

shell(+CmdSpec,-StreamToProc, -StreamFromProc, -ProcStderr, -ErrorCode)

The arguments of this predicate are similar to those of spawn_process, except for the following: (1) The first argument is an atom or a list of atoms, like in spawn_process. However, if it is a list of atoms, then the resulting shell command is obtained by string concatenation. This is different from spawn_process where each member of the list must represent an argument to the program being invoked (and which must be the first member of that list). (2) The last argument is the error code returned by the shell command and not a process id. The code -1 and 127 mean that the shell command failed.

The shell/5 predicate is similar to spawn_process in that it spawns another process and can capture that process' input and output streams. The important difference, however, is that XSB will wait until the process spawned by shell/5 terminates. In contrast, the process spawned by spawn_process will run concurrently with XSB. In this latter case, XSB must explicitly synchronize with the spawned subprocess using the predicate process_control/2 (using the wait operation), as described earlier.

The fact that XSB must wait until shell/5 finishes has a very important implication: the amount of data the can be sent to and from the shell command is limited (1K is probably safe). This is because the shell command communicates with XSB via pipes, which have limited capacity. So, if the pipe is filled, XSB will hang waiting for shell/5 to finish and shell/5 will wait for XSB to consume data from the pipe. Thus, use spawn_process/5 for any kind of significant data exchange between external processes and XSB.

```
| ?- [Compiling ./child]              <- The child copy of received the pipe from
[child compiled, cpu time used: 0.1300 seconds]      the parent and then the
[child loaded]                                       request to consult child.P
Pipe 15 received                    <- child.P acknowledges receipt of the pipe
Message was: Hello!                 <- child.P gets the message and prints it
yes
```

Observe that the parent process is very careful about making sure that the child terminates and also about closing the I/O streams after they are no longer needed.

Finally, we should note that this mechanism can be used to communicate through pipes with non-XSB processes as well. Indeed, an XSB process can create a pipe using pipe_open (before spawning a child process), pass one end of the pipe to a child process (which can be a C program), and use open/3 to convert the other end of the pipe to an XSB stream. The C program, of course, does not need open/3, since it can use the pipe file handle directly. Likewise, a C program can spawn off an XSB process and pass it one end of a pipe. The XSB child-process can then convert this pipe fd to a file using fd2iostream and then talk to the parent C program.

fd2iostream(+Pipe, -IOstream)

Take a file descriptor and convert it to an XSB I/O stream. This predicate should be used only for user-defined I/O. Otherwise, use open/{3,4} when possible.

1.7 Socket I/O

The XSB socket library defines a number of predicates for communication over BSD-style sockets. Most are modeled after and are interfaces to the socket functions with the same name. For detailed information on sockets, the reader is referred to the Unix man pages (another good source is Unix Network Programming, by W. Richard Stevens). Several examples of the use of the XSB sockets interface can be found in the XSB/examples/ directory in the XSB distribution.

XSB supports two modes of communication via sockets: stream-oriented and message-oriented. In turn, stream-oriented communication can be buffered or character-at-a-time.

To use buffered stream-oriented communication, system socket handles must be converted to XSB I/O streams using fd2iostream/2. In these stream-oriented communication, messages have no boundaries, and communication appears to the processes as reading and writing to a file. At present, buffered stream-oriented communication works under Unix only.

Character-at-a-time stream communication is accomplished using the primitives socket_put/3 and socket_get0/3. These correspond to the usual Prolog put/1 and get0/1 I/O primitives.

In message-oriented communication, processes exchange messages that have well-defined boundaries. The communicating processes use socket_send/3 and socket_recv/3 to talk to each other. XSB messages are represented as strings where the first four bytes (sizeof(int)) is an integer (represented in the binary network format: see the functions htonl and ntohl in socket docu-

mentation) and the rest is the body of the message. The integer in the header represents the length of the message body.

Effort has been made to make the socket interface thread-safe, however in Version 3.5, calls to the XSB socket interface go through a single mutex, and may cause contention if many threads seek to concurrently use sockets.

We now describe the XSB socket interface. All predicates below must be imported from the module socket. Note that almost all predicates have the last argument that unifies with the error code returned from the corresponding socket operation. This argument is explained separately.

General socket calls. These are used to open/close sockets, to establish connections, and set special socket options.

socket(-Sockfd, ?ErrorCode)
: A socket Sockfd in the AF_INET domain is created. (The AF_UNIX domain is not yet implemented). Sockfd is bound to a small integer, called socket descriptor or socket handle.

socket_set_option(+Sockfd,+OptionName,+Value)
: Set socket option. At present, only the linger option is supported. Lingering is a situation when a socket continues to live after it was shut down by the owner. This is used in order to let the client program that uses the socket to finish reading or writing from/to the socket. Value represents the number of seconds to linger. The value -1 means do not linger at all.

socket_close(+Sockfd, ?ErrorCode)
: Sockfd is closed. Sockets used in socket_connect/2 should not be closed by socket_close/1 as they will be closed when the corresponding stream is closed.

socket_bind(+Sockfd,+Port, ?ErrorCode)
: The socket Sockfd is bound to the specified local port number.

socket_connect(+Sockfd,+Port,+Hostname,?ErrorCode)
: The socket Sockfd is connected to the address (Hostname and Port). If socket_connect/4 terminates abnormally for any reason (connection refused, timeout, etc.), then XSB closes the socket Sockfd automatically, because such a socket cannot be used according to the BSD semantics. Therefore, it is always a good idea to check to the return code and reopen the socket, if the error code is not SOCK_OK.

socket_listen(+Socket, +Length, ?ErrorCode)
: The socket Sockfd is defined to have a maximum backlog queue of Length pending connections.

socket_accept(+Sockfd,-SockOut, ?ErrorCode)
: Block the caller until a connection attempt arrives. If the incoming queue is not empty, the first connection request is accepted, the call succeeds and returns a new socket, SockOut, which can be used for this new connection.

set_scan_pars(+List) module: stdscan

set_scan_pars(+List) is used to con gure the tokenizer to a particular need. List is a list of parameters including the following:

whitespace . The default action of the scanner is to return a list of tokens, with any whitespace removed. If whitespace is a parameter, then the scanner returns the token " when it nds whitespace separating two tokens (unless the two tokens are letter sequences; since two letter sequences can be two tokens ONLY if they are separated by whitespace, such an indication of whitespace would be redundant.) Including the parameter no_whitespace undoes the e ect of previously including whitespace .

upper_case The default action of the parser is to treat lowercase letter di erently from uppercase letters. This parameter should be set if conversion to uppercase should be done when producing a token that does not consist entirely of letters (e.g. one with mixed letters and digits). Including the parameter no_case undoes the e ect of previously including upper_case.

upper_case_in_lit The default action of the parser is to treat lowercase letter di erently from uppercase letters. This parameter should be set if conversion to uppercase should be done when producing a token that consists entirely of letters. Including the parameter no_case_in_lit undoes the e ect of previously including upper_case.

whitespace(Code) adds Code as a whitespace code. By default, all ASCII codes less than or equal to 32 are regarded as whitespace.

letter(Code) adds Code as a letter constituting a token. By default, ASCII codes for characters a z and A Z are regarded as letters.

special_char(Code) adds Code as a special character. By default, ASCII codes for the following characters are regarded as special characters:

| { } [ ] " % $ & ' ( ) * + , - . / : ; < = > ? @ \ ^ _ ~ `

get_scan_pars(-List) module: stdscan

get_scan_pars/1 returns a list of the currently active parameters.

1.14 XSB Lint

The xsb_lint_impexp.P le contains a simple tool to analyze import/exports and de nitions and uses of predicates. It tries to nd possible inconsistencies, producing warnings when it nds them and generating document_import / document_export declarations that might be useful. It can be used after a large multi- le, multi-module XSB program has been written to nd possible inconsistencies in (or interesting aspects of) how predicates are de ned and used.

XSB source les that contain an export compiler directive are considered as modules. Predicates de ned in modules, but not exported, are local to that module. When compiling a module, the XSB compiler generates useful warnings when predicates are used but not de ned or de ned

called with ClauseHead as a variable.) Note also that the order of clauses is not preserved and is indeterminate.

call_in_db(?Goal,+DB)
: call_in_db/2 calls Goal in DB and returns all instances of Goal provable by rules in DB. Clauses must not contain cuts (!). They can contain most Prolog constructs, including and, or, if-then-else, \+, calls to standard predicates, and calls explicitly modified by a module name. Such calls will be satisfied by calling the goal in the indicated module. So in this case one can think of a Prolog DB as being extended by the code in any module.

load_in_db(+FileName,+DB0,-DB)
: load_in_db/3 reads the clauses from the file named FileName and asserts them into database DB0 returning DB.

load_in_db(+FileName,-DB)
: load_in_db/2 reads the clauses from the file named FileName and asserts them into an empty database returning DB.

union_db(+DB1,+DB2,-DB3)
: union_db/3 returns in DB3 the union of the sets of clauses in DB1 and DB2.

1.16 Miscellaneous Predicates

term_hash(+Term,+HashSize,-HashVal) module: machine
: Given an arbitrary Prolog term, Term, that is to be hashed into a table of HashSize buckets, this predicate returns a hash value for Term that is between 0 and HashSize -1.

pretty_print(+ClausePairs) module: pretty_print

pretty_print(+Stream,+ClausePairs) module: pretty_print
: The input to pretty_print/1, ClausePairs, can be either a list of clause pairs or a single clause pair. A clause pair is either a Prolog clause (or declaration) or a pair:

(Clause,Dict)

Where Dict is a list of the form A = V where V is a variable in Clause and A is the string to be used to denote the variable[12].

By default, pretty_print/1 outputs atomic terms using writeq/1, but specialized output can be configured via asserting in usermod a term of the form

user_replacement_hook(Term,Call)

which will use Call to output an atomic literal A whenever A unifies with Term. For example, pretty printing weight constraints in XSB's XASP package is done via the hook

[12] Thus the list of variable names returned by read_term/{2,3} can be used directly in Dict.

gensym(+Atom,-Var) module: gensym

Generates a new integer, and concatenates this integer with Atom, unifying the result with Var. For instance a call ?- gensym(foo,Var) might unify Var with foo32.

1.11 Random Number Generator

The following predicates are provided in module random to generate random numbers (both integers and oating numbers), based on the Wichmann-Hill Algorithm [30, 19]. The random number generator is entirely portable, and does not require any calls to the operating system. As noted below, it does require 3 seeds, each of which must be an integer in a given range. These seeds are thread-speci c: thus di erent threads may generate independent sequences of random numbers.

random(-Number) module: random

Binds Number to a random oat in the interval [0.0, 1.0). Note that 1.0 will never be generated.

random(+Lower,+Upper,-Number) module: random

Binds Number to a random integer in the interval [Lower,Upper) if Lower and Upper are integers. Otherwise Number is bound to a random oat between Lower and Upper. Upper will never be generated.

getrand(?State) module: random

Tries to unify State with the term rand(X,Y,Z) where X,Y, and Z are integers describing the state of the random generator.

setrand(rand(+X,+Y,+Z)) module: random

Sets the state of the random generator. X,Y, and Z must be integers in the ranges [1,30269), [1,30307), [1,30323), respectively.

datime_setrand module: random

This simple initialization utility sets the random seed triple based on a function of the current day, hour, minute and second.

randseq(+K, +N, -RandomSeq) module: random

Generates a sequence of K unique integers chosen randomly in the range from 1 to N. RandomSeq is not returned in any particular order.

randset(+K, +N, -RandomSet) module: random

Generates an ordered set of K unique integers chosen randomly in the range from 1 to N. The set is returned in reversed order, with the largest element rst and the smallest last.

gauss(-G1,-G2) module: ramdom

Generates two random numbers that are normally distributed with mean 0 and standard deviation 1. It uses the polar form of the Box-Muller transformation [5] of uniform random variables as generated by random/1.

```
| ?- 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 , rel 1.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 de ned 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 rst 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 nd an unused cursor and retrieve the results. In this way e cient 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).
```

2.2.8 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 rst argument is the declared database predicate for insertions

argument position of a use of a meta-predicate, are considered uses of that predicate symbol. The default is also to allow nonconstant symbols appearing in any other context to also count as uses of that symbol as that predicate symbol. This is useful for programs that de ne their own meta-predicates.

checkImpExps(+FileNameList)
checkImpExps/1 is currently equivalent to checkImpExps([],FileNameList) .

1.15 Pure Meta-programming in XSB with prolog_db.P

The prolog_db library provides predicates that support a form of pure meta-programming in XSB. A programmer can create a term data structure that represents a Prolog database (i.e., a set of rules, and herein called a Prolog DB), and then ask for a goal to be proved in such a Prolog DB.

A Prolog DB is kept as a trie, which is a ground Prolog term. Each level in the trie is implemented by a hash table, and hash tables are expanded and contracted as necessary. A set of clauses is canonically represented, i.e., no matter what sequence of assert_in_db 's and retractall_in_db 's one uses to construct a particular set of clauses, the resulting Prolog DBs (i.e. Prolog terms) are identical.

A Prolog DB represents an unordered set of clauses. The order in which clauses are returned from clause_in_db (and thus for call_in_db) is indeterminate, and may change from one call to the next (due to possible expansion or contraction of a hash table in the representation of a Prolog DB.)

A Prolog DB that is obtained from another Prolog DB by adding or deleting a single clause di ers from it in only log subterms (unless a hash table has been resized). This means that it is e cient to intern these DB's, and to table them (as intern).

The predicates provided by the Prolog DB interface are as follows:

empty_db(-EmptyPrologDB)
empty_db/1 returns an empty Prolog DB. It is used to create an initial Prolog DB to pass to the other in_db predicates.

assert_in_db(+Clause,+DB0,-DB)
assert_in_db/3 adds the clause, Clause, to the Prolog DB, DB0, and returns a new Prolog DB, DB A Prolog DB is a set of clauses, so asserting a clause that is already in DB0 just returns that same database. No ordering of clauses is preserved, so cuts do not make sense and cannot be used in clauses. (The if-then else ('->'/3) should be used instead.)

retractall_in_db(+ClauseHead,+DB0,-DB)
retractall_in_db/3 removes all clauses whose heads unify with ClauseHead from DB0 returning DB If no clauses in DB0 unify, then DB0 is returned unchanged.

clause_in_db(?ClauseHead,?ClauseBody,+DB)
clause_in_db/3 returns all clauses in DB whose heads and bodies unify with ClauseHead and ClauseBody, respectively. (Note that, unlike clause/2 in Prolog, clause_in_db can be

```
user_replacement_hook(weight_constr(Term),output_weight_constr(Term))
```

which outputs a weight constraint in a (non-Prolog) syntax that is used by several ASP systems.

module_of_term(+Term,?Module) module: machine
Given a term Term, module_of_term/2 returns the module of its main functor symbol in Module. If the module cannot be determined wither unknown1 or unknown2 is returrned, depending on the reason the module name cannot be determined.

1.17 Other Libraries

Not all XSB libraries are fully documented. We provide brief summaries of some of these other libraries.

1.17.1 Justi cation

By Hai-Feng Guo

Most Prolog debuggers, including XSB's, are based on a mechanism that allows a user to trace the evaluation of a goal by interrupting the evaluation at call, success, retry, or failure of various subgoals. While this has proved an excellent mechanism for evaluating SLD(NF) executions, it is di cult at best to use such a mechanism during a tabled evaluation. This is because, unlike with SLD(NF), SLG requires answers to be returned to tabled subgoals at various times (depending on whether batched or local evaluation is used), negative subgoals to be sometimes be delayed and/or simpli ed, etc.

One approach to understanding tabled evaluation better is to abstract away the procedural aspects of debugging and to use the tables produced by an evaluation to construct justi cation after the evaluation has nished. The justi cation library does just this using algorithms described in [14].

1.17.2 AVL Trees

By Mats Carlsson

AVL trees provide a mechanism to maintain key value pairs so that loop up, insertion, and deletion all have complexity $O(\log n)$. This library contains predicates to transform a sorted list to an AVL tree and back, along with predicates to manipulate the AVL trees.

1.17.3 Ordered Sets: ordsets.P

By Richard O'Keefe

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.

11.6.4 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 p        assive(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,O     ldSkel)),
            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 speci es 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 nal 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 in nite 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.

11.6.5 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 speci ed CHR module, in the format of an answer store usable as a return argument of a tabled predicate.

11.7 Guidelines

In this section we cover several guidelines on how to use CHR to write constraint solvers and how to do so e ciently.

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 a ects e ciency and possibly termination

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'

2.2.11 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 o , 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(o)) Turns o 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 e ect if autocommit is o).

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

2.2.12 Interface Flags

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

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

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

     Val = on
```

Chapter 3

The New XSB-Database Interface

By Saikat Mukherjee, Michael Kifer and Hui Wan

3.1 Introduction

The XSB-DB interface is a package that allows XSB users to access databases through various drivers. Using this interface, information in di erent DBMSs can be accessed by SQL queries. The interface de nes 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 di erent databases) and keep alive multiple queries (again from the same or di erent connections). The interface also supports dynamic loading of drivers. As a result, it is possible to query databases using di erent drivers concurrently[1].

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

3.2 Con guring the Interface

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

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

```
makexsb_wind
```

[1] In Version 3.5, this package has not been ported to the multi-threaded engine.

XSB_DBI_010: XSB_DBI ERROR: Same query handle used for di erent 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: XSB_DBI ERROR: Number of requested columns exceeds the number of columns in the query
This error is thrown when the user db_query speci es 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: XSB_DBI ERROR: Number of requested columns is less than the number of columns in the query
This error is thrown when the user db_query speci es 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: XSB_DBI ERROR: Invalid return list in query
Something else is wrong with the return list of the query.

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

XSB_DBI_015: XSB_DBI ERROR: 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: XSB_DBI ERROR: Too many active queries
There is a limit (2000) on the number of queries that can remain open at any given time.

3.5 Notes on speci c drivers

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.

Chapter 8

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

By Rohan Shirwaikar

8.1 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 CVS, you need to compile the package as follows:

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

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 8.4.

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