In the Lecture Series Logic Programming and Constraints

Non Logical

Presented by Stéphane Bressan

References

- Chapters 9, 10, 11 and 21 of the user manual
- Self-study chapter 20

Global Data Structures

Normal Logic Programming does not allow to share information across parallel branches of the SLD tree and to get information from failing branches.

To do so, we need global data structures.

<table>
<thead>
<tr>
<th>Facility</th>
<th>Type</th>
<th>Access</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>shelves</td>
<td>array</td>
<td>by handle</td>
<td>shelf_create/2,3</td>
</tr>
<tr>
<td>bags</td>
<td>unordered bag</td>
<td>by handle</td>
<td>bag_create/1</td>
</tr>
<tr>
<td>stores</td>
<td>hash table</td>
<td>by handle</td>
<td>store_create/1</td>
</tr>
<tr>
<td>named shelves</td>
<td>array</td>
<td>by name</td>
<td>shelf/2</td>
</tr>
<tr>
<td>named stores</td>
<td>hash table</td>
<td>by name</td>
<td>store/1</td>
</tr>
<tr>
<td>non-logical variables</td>
<td>single cell</td>
<td>by name</td>
<td>variable/1</td>
</tr>
<tr>
<td>non-logical arrays</td>
<td>array</td>
<td>by name</td>
<td>array/1,2</td>
</tr>
<tr>
<td>records</td>
<td>ordered list</td>
<td>by name</td>
<td>record/1,2</td>
</tr>
<tr>
<td>dynamic predicates</td>
<td>ordered list</td>
<td>by name</td>
<td>dynamic/1,assert/1</td>
</tr>
</tbody>
</table>

Non-logical Variables

Non-logical variables are global (to a module) named variables (an atom) with destructive assignment.

- It is good style to declare variables
  - :- local variable(count).
- A value, a term, is assigned to the variable
  - setval(+ElemSpec, ?Value)
- The value is retrieved
  - getval(+ElemSpec, ?Value)

Non-logical Variables

If the value is a non ground term, the retrieved value is a variant (i.e. different variables)

:- V = f(X, Y, X),
setval(test, V),
getval(test, V1),
getval(test, V2),
writeq([V, V1, V2]).
Non-logical Variables

```prolog
count_solutions(Goal, _) :-
    setval(count, 0),
call(Goal),
incval(count),
fail.
count_solutions(_, N) :-
    getval(count, N).
```

Bags

```prolog
count_solutions(Goal, _) :-
    setval(count, 0),
call(Goal),
incval(count),
fail.
count_solutions(_, N) :-
    getval(count, N).
```

```prolog
bags

simple_findall(Goal, Solutions) :-
    bag_create(Bag),
    ( call(Goal),
      bag_enter(Bag, Goal),
      fail ;
      true ),
    bag_dissolve(Bag, Solutions).
```

Dynamic Predicates

```prolog
:- dynamic a/2.
:- assert(a(c, Y)).
:- listing a/2.
:- assert(a(c, d)).
:- assert(a(X, Y) :- writeln("hello")).
:- a(X, Y).
:- listing a/2.
:- retract(a(X, Y)).
:- listing a/2.
:- retract(H:-B).
:- retract(a(X, Y):-B).
:- listing a/2.
```

Input and Output

```prolog
:- dynamic a/2.
:- assert(a(c, Y)).
:- listing a/2.
:- assert(a(c, d)).
:- assert(a(X, Y) :- writeln("hello")).
:- a(X, Y).
:- listing a/2.
:- retract(a(X, Y)).
:- listing a/2.
:- retract(H:-B).
:- retract(a(X, Y):-B).
:- listing a/2.
```
Input and Output

- Input: 0 (stdin)
- Output: 1 (stdout)
- warning_output: 1 (stdout)
- log_output: 1 (stdout)
- error: 2 (stderr)
- null 3 (null)
- User: 0 or 1 (stdin or stdout)

Input and Output

See also: tyi/1, tyi/2, tyo/1, tyo/2, read_string/3, read_string/4, readvar/3, read_term/2, read_term/3, display / 1, display / 2, writeq/1, writeq/2, write_canonical/1, write_canonical/2, nl/0, nl/1, writeln/1, writeln/2, print/1, print/2, printf/2, printf/3, at_eof/1 and set_stream_property/3.

Input and Output

- open(++)Source, +Mode, ?Stream)
  - Opens the I/O source or sink Source in mode Mode and associates it with the stream identifier Stream.
  - SourceSink can be
    - a file (path and name),
    - a string (string(String)), or
    - a queue (queue(String))
  - Mode can be one of
    - read,
    - write,
    - update,
    - Append
  - close(+)Stream)
    - Closes the stream specified by Stream.

Input and Output

- get(+Stream, -Ascii), get(-Ascii)
  - Reads the next character from the input stream Stream and unifies its ASCII code with Ascii.
- put(+Stream, +Ascii)
  - The character represented by the ascii integer code AAscii is put onto the buffered output stream Stream.
- read_token(+Stream, -Token, -Class)
  - Succeeds if the next token from the input stream Stream is successfully read and unified with Token and its token class with Class.
- read(+Stream, -Term)
  - Succeeds if the next term from the input stream Stream is successfully read and unified with Term.
- write(+Stream, ?Term)
  - The term Term is written on the output stream Stream according to the current operator declarations.
- seek(+Stream, +Offset)
  - The pointer in stream Stream is offset Offset from the start of the file.
- at(+Stream, -Pointer)
  - Succeeds if Position is the position of the stream Stream.
- pipe(?StreamIn, ?StreamOut)
  - Creates a pipe and two streams StreamIn and StreamOut to its read and write ends.

Input and Output

- socket(+Domain, +Type, ?Stream)
  - Creates a socket of a given type and domain and associates a stream with it.
  - Domain is unix or internet.
  - Type is stream or datagram.
- accept(+Stream, -From, ?NewStream)
  - Accepts a connection for a stream socket and creates a new socket which can be used for I/O.

See also bind / 2, listen / 2, connect / 2

Internet and Inter-process Communication

- Socket Domains
- Stream Connection (internet domain)
- Datagram Connection (internet domain)
- Stream Connection (unix domain)
- Datagram Connection (unix domain)
Internet Communication

server:
[predicate 10]: socket(internet, stream, s), bind(s, X).
X = m1 / 3789
yes.
[predicate 11]: listen(s, 1), accept(s, From, news).
<blocks waiting for a connection>
client:
[predicate 20]: socket(internet, stream, s), connect(s, m1/3789).
yes.
[predicate 21]: printf(s, "Msg: "%s", message(client)), read(s, Msg).
server:
From = m2 / 1627
yes.
[predicate 12]: read(news, Msg), printf(news, "Msg: "%s", message(server)).
Msg = message(client)
yes.
client:
Msg = message(server)
yes.