TCPW/CW/K.INT.

BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

Interface between Prolog and Unify : Higher Level

by
Jose COTTA *
Raf VENKEN *

Internal Report
BIM-prolog IR7

October 1984

* BIM
Kwikstraat 4
B-3078 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
B-3030 Heverlee Belgium
tel. +32 16 20 06 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

INTERFACE BETWEEN PROLOG AND UNIFY: HIGHER LEVEL

by

Jose Cotta
Raf Venken
Belgian Institute of Management
Kwikstraat 4
3078 Everberg
Belgium

Contents

1. Introduction

In this report we specify the interface between the **Prolog** system and the relational database management system **Unify**.

As it is stated in <Ven84> there are different levels of interface, namely an higher level where the database system has to answer to joins of database calls and to individual relation calls, and a lower level where Prolog accesses the database tuple by tuple by means of "seek", "getnext" and similar procedures.

As far as this report is concerned we shall only specify the higher level of interface. The lower level will soon have its own specification report.

This higher level interface is based on the following principles:

1) There exists a **pipe-line** communication channel between Prolog and a Scheduler of the database accesses:

```
--------------                          ---------------
|            |    database calls       |             |
|            |   ---------------->     |             |
|            |                         |             |
|   Prolog   |   database answers      |  Scheduler  |
|            |   <----------------     |             |
|            |                         |             |
|            |                         |             |
--------------                          ---------------
```

The scheduler will control the database system in a completely transparent way from the Prolog point of vue.

2) To the Prolog system, it only matters that it sends a **database question** through the pipe-line to the Scheduler and that it gets back a **database answer** through the same pipe-line.

There are nine different database questions: opendb, closedb, retrieve, createrelation, insert, delete, deleterelation, backtracking and cut that will be described in this report.

3) The filosophy of the interface is based on an uniform type of communication through the pipe-line, ie, Prolog sends a database question accordingly to a well defined syntax, we will call it the **ibuffer** from now on, and receives a database

answer also accordingly to a well defined syntax, we will call it **obuffer** from now on.

Therefore, the aim of this paper is, on one hand, to specify the syntax of the ibuffer and obuffer and, on the other hand, to describe the actions that must be performed for each of the database questions.

2. Syntax of database questions and answers

2.1 ibuffer

All the database questions will be sent by the Prolog system to the Scheduler, through the pipe-line, accordingly to the following syntax, stated in BNF form:

```
<ibuffer> ::= <code> <arity of the code> [ <iblock sequence> ]

<code> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<arity of the code> ::= 0 | positive integer

<iblock sequence> ::= <iblock> [ <iblock sequence> ]

<iblock> ::= <arity of the iblock> <smallblock sequence>

<arity of the iblock> ::= positive integer

<smallblock sequence> ::= <smallblock> [ <smallblock sequence> ]

<smallblock> ::= <length> <type of sequence> <byte sequence>

<length> ::= positive integer

<type of sequence> ::= 1 | 2 | 3 | 4

<byte sequence> ::= byte [ <byte sequence> ]
```

This syntax suggests the following comments:

1) <code> represents the code of the action to be performed; the <arity of the code> is the number of <iblock>'s that are inside its <iblock sequence>.

2) Each <iblock> has also an <arity of the iblock> that specifies the number of <small block>'s that are inside its <small block sequence>.

3) Each <small block> has a certain <length> in bytes, a certain <type of sequence> and a <byte sequence>; the <type of sequence> has the value 1 for character strings, 2 for integer numbers, 3 for variables, and 4 for void variables (which are variables whose values are not important for the Prolog system).

4) Therefore an <iblock sequence> can only be empty when the <arity of the code> is 0. At present, as we will see in section 3, only one code has arity 0.

5) A <smallblock sequence> and a <byte sequence> can never be empty.

In section 3 we will give examples of the ibuffers generated by the several database questions.

2.2 obuffer

All the database answers will be sent by the Scheduler to the Prolog system, through the pipe-line, accordingly to the following syntax, stated also in BNF form:

```
<obuffer> ::= <result> <relative identification>
              <last solution> <number of oblocks>
              [ <oblock sequence> ]

<result> ::= 0 | negative integer

<relative identification> ::= positive integer

<last solution> ::= 0 | 1

<number of blocks> ::= 0 | positive integer

<oblock sequence> ::= <oblock> [ <oblock sequence> ]

<oblock> ::= <length> <variable number> <type of sequence>
             <byte sequence>

<length> ::= positive integer

<variable number> ::= negative integer

<type of sequence> ::= 1 | 2

<byte sequence> ::= byte [ <byte sequence> ]
```

This syntax suggests the following comments:

1) <result> can be 0 meaning successful operation or an error code represented by a negative integer in case of unsuccessful operation.

2) <relative identification> is a number that identifies biunivocally the operation performed. It may be needed later for the backtracking of Prolog.

3) <last solution> is 0 if there are other solutions to the query or 1 if the Scheduler is sure that the present solution is the last one.

4) The <number of blocks> can be 0 if there is no need to send information back to the Prolog system (for example in an insert operation).

5) Each <oblock> represents a variable, in the Prolog sense, that has been instantiated during the operation, therefore it must have its <variable number>.

6) The <type of sequence> can be 1 or 2 if the sequence is a string or an integer number, respectively.

Note that the <oblock> and the <iblock> have completely different syntax rules. In the next section we will give examples of the obuffers expected by the Prolog system as answers to the several database questions.

3. Description of the several database questions

Each of the database questions generates a different ibuffer and Prolog expects, also different obuffers as answers to them. Therefore, in this section we will describe in detail all the database questions and the operations that must be executed by the database management system for each of them. We will also give a lot of examples to illustrate the description.

3.1 Opendb

This question opens a specified database for further use. Therefore, it must be called always in the beginning of the session, whether we want to create a new database or to use an already existing one. Its syntax is the following:

<code> = 1;
<arity of the code> = 2;
the first <iblock> specifies the name of the database that is going to be used; and
the second <iblock> contains a list of options for the database opening.

The list of options is a four digit integer 'nnnn' composed in the following way:

the first digit represents read field names (1=yes ; 0=no);
the second digit represents read field synonyms (1=yes ; 0=no);
the third digit represents read record names (1=yes ; 0=no); and
the fourth digit represents read record synonyms (1=yes ; 0=no).

Let's see how this operation is called through some examples.

Example 3.1.1: (New database)

Suppose that we want to create a database called "mydata". The ibuffer for this question would be:

```
----------------------------------------
|  1  |  code
|-----+------------------------
|  2  |  arity of the code
|-----+------------------------
|  1  |  arity of the iblock
|-----+---------------------
|  7  |  length
|-----+--------
|  1  |  type
|-----+--------
|  m  |
|  y  |
|  d  |
|  a  |
|  t  |
|  a  |
|-----+-----------------------
|  1  | arity of the iblock
|-----+---------------------
|  2  | length
|-----+--------
|  2  | type
|-----+-------
|  0  |
----------------------------------------
```

In this example the last 0 stands for 0000 because it is a new database. Assuming that the operation was sucessful and that its relative identification number was 1, the obuffer sent to Prolog would be:

```
----------------------------------------
|  0  | result
|-----+-------------------------
|  1  | relative identification
|-----+-------------------------
|  1  | last solution
|-----+-------------------------
|  0  | number of oblocks
----------------------------------------
```

Example 3.1.2: (Already existing database)

Let's now suppose that the database "mydata" already exists and that its schema has record and field names. The ibuffer to call such an opening would be:

```
----------------------------------------
|  1  | code
|-----+----------------------
|  2  | arity of the code
|-----+----------------------
|  1  | arity of the iblock
|-----+---------------------
|  7  | length
|-----+--------
|  1  | type
|-----+--------
|  m  |
|  y  |
|  d  |
|  a  |
|  t  |
|  a  |
|-----+------------------------
|  1  |  arity of the iblock
|-----+---------------------
|  2  |  length
|-----+--------
|  2  |  type
|-----+--------
| 1010|
---------------------------------------
```

Assuming that the operation was sucessful and that its relative identification number was 1, the obuffer sent to Prolog would be:

```
---------------------------------------
|  0  |  result
|-----+-------------------------
|  1  |  relative identification
|-----+-------------------------
|  1  |  last solution
|-----+-------------------------
|  0  |  number of oblocks
---------------------------------------
```

Notes about the opendb operation:

1) This operation must preceed any of the other operations.

2) When the operation is sucessful the database is opened accordingly to the list of options.

3) The operation is not sucessful, and therefore the database is not opened, when it is already opened or when the list of options can not be satisfied. In this case the error code will be transmited in <result>.

3.2 Closedb

This question closes the currently opened database. Therefore, it must be the last call of the session. If a "closedb" is called before the end of the session, the other existing questions can't be executed. Its syntax is the following:

<code> = 2; and
<arity of the code> = 0.

Example 3.2.1:

If the Prolog system wants to close the currently opened database it will send the following ibuffer:

2	code
0	arity of the code

Assuming that the operation was sucessful and that its relative identification number was 1, the answer of the Scheduler would be:

0	result
1	relative identification
1	last solution
0	number of oblocks

Notes about the closedb operation:

1) This operation must be called in the end of the session.

2) This operation is not sucessful if there is no database currently opened. Otherwise it is always sucessful.

3.3 Retrieve

There are two types of retrieve: in the first one we want to retrieve a certain relation table and in the second one we want to access a tuple of a join of relations.

The syntax of this database question is the same in the two above cases:

<code> = 3;
<arity of the code> = 2;
the first <iblock> specifies the names of the relations that are involved in the database question; and
the second <iblock> contains the information supplied by Prolog in order that the operation might be performed.

Example 3.3.1: (Retrieve without variables)

Let us suppose that we have a database with facts about persons and their addresses, thus the following table:

ADDRESS/2

NAME	PLACE
Raf	Leuven
Jose	Heverlee

Which is represented, in Prolog, by:

```
address(Raf,Leuven).
address(Jose,Heverlee).
```

The Prolog query: ?-address(Raf,Leuven). (is Leuven the address of Raf?) would generate a database question, and the following ibuffer would be sent to the Scheduler, through the pipe-line:

3	code
2	arity of the code
1	arity of the iblock
10	length
1	type
a d d r e s s / 2	
2	arity of the iblock
4	length
1	type

```
|  R  |
|  a  |
|  f  |
|-----+--------------------
|  7  |  length
|-----+--------
|  1  |  type
|-----+--------
|  L  |
|  e  |
|  u  |
|  v  |
|  e  |
|  n  |
----------------------------------------
```

Assuming that the relative identification of this operation was 1, the obuffer that Prolog would expect, would be:

```
----------------------------------------
|  0  |  result
|-----+-------------------------
|  1  |  relative identification
|-----+-------------------------
|  1  |  last solution
|-----+-------------------------
|  0  |  number of oblocks
----------------------------------------
```

Example 3.3.2: (Retrieve with one variable)

In the same database, the Prolog query: "?-address(Jose,_X)." (where does Jose live?) would generate the following ibuffer to be sent to the Scheduler:

```
----------------------------------------
|  3  |  code
|-----+---------------------------
|  2  |  arity of the code
|-----+---------------------------
|  1  |  arity of the iblock
|-----+---------------------
| 10  |  length
|-----+--------
|  1  |  type
|-----+--------
|  a  |
|  d  |
|  d  |
|  r  |
|  e  |
|  s  |
|  s  |
|  /  |
|  2  |
|-----+-------------------------
|  2  |  arity of the iblock
|-----+--------------------
|  5  |  length
|-----+--------
|  1  |  type
|-----+--------
|  J  |
|  o  |
|  s  |
|  e  |
|-----+--------------
|  2  |  length
|-----+--------
|  3  |  type
|-----+--------
| -1  |
---------------------------------------
```

In the last <smallblock> the byte -1 represents the variable X of the Prolog query. It's value is thus the aim of the query. Therefore the obuffer for this query would be:

```
---------------------------------------
|  0  |  result
|-----+-----------------------
|  1  |  relative identification
|-----+-----------------------
|  1  |  last solution
|-----+-----------------------
|  1  |  number of oblocks
|-----+-----------------------
| 10  |  length
|-----+----------------
| -1  |  variable number
|-----+----------------
|  1  |  type
|-----+----------------
|  H  |
|  e  |
|  v  |
|  e  |
|  r  |
|  l  |
|  e  |
|  e  |
---------------------------------------
```

Example 3.3.3: (retrieve with a void variable)

Let us now suppose that in the former query Prolog was not interested in the value of the variable X. This means that the query would be: has Jose an address? and it could be written in Prolog syntax: "?-address(Jose,_)." .

This difference would generate a ibuffer just like the previous one but with the last <smallblock> modified in the following way:

1	length
4	type

The type 4 means a void variable, therefore the obuffer with the answer just has to mention the result, the last solution with the value 1, and the relative identification of the process.

0	result
1	relative identification
1	last solution
0	number of iblocks

Example 3.3.4: (Retrieve of a join of relations)

Let us now suppose that in the previous database we have also a table for the relation 'age':

AGE/2

name	age
Raf	26
Jose	28

The Prolog query: "?-age(Raf,_X),address(Raf,_Y)." (how old is Raf and where does he live?) is in fact a join of two database calls, so its ibuffer will have in the first iblock the two relation names and in the second iblock their arguments, as follows:

```
-----------------------------------------
|  3  | code
|-----+-------------------------
|  2  | arity of the code
|-----+-------------------------
|  2  | arity of the iblock
|-----+--------------------
|  6  | length
|-----+--------
|  1  | type
|-----+--------
|  a  |
|  g  |
|  e  |
|  /  |
|  2  |
|-----+---------------
| 10  | length
|-----+--------
|  1  | type
|-----+--------
|  a  |
|  d  |
|  d  |
|  r  |
|  e  |
|  s  |
|  s  |
|  /  |
|  2  |
|-----+-------------------------
|  4  | arity of the iblock
|-----+--------------------
|  4  | length
|-----+--------
|  1  | type
|-----+--------
|  R  |
|  a  |
|  f  |
|-----+--------------------
|  2  | length
|-----+--------
|  3  | type
|-----+--------
| -1  |
|-----+--------------------
|  4  | length
```

```
|-----+--------
|  1  |  type
|-----+--------
|  R  |
|  a  |
|  f  |
|-----+---------------------
|  2  |  length
|-----+--------
|  3  |  type
|-----+--------
| -2  |
----------------------------------------
```

The obuffer for this query would be:

```
----------------------------------------
|  0  |  result
|-----+-------------------------
|  1  |  relative identification
|-----+-------------------------
|  1  |  last solution
|-----+-------------------------
|  2  |  number of oblocks
|-----+-------------------------
|  3  |  length
|-----+-----------------
| -1  |  variable number
|-----+-----------------
|  2  |  type
|-----+-----------------
| 26  |
|-----+-------------------------
|  8  |  length
|-----+-----------------
| -2  |  variable number
|-----+-----------------
|  1  |  type
|-----+-----------------
|  L  |
|  e  |
|  u  |
|  v  |
|  e  |
|  n  |
----------------------------------------
```

Notes about the retrieve operation:

1) The contents of the database remains unchanged after a retrieve operation. This operation just accesses the

information stored in the database.

2) The variable numbers that appear in obuffer are the ones that were sent to the Scheduler in the previous ibuffer.

3) When the retrieve represents a join of database calls the small blocks that appears inside the second iblock are the arguments of the relations involved and they appear ordered.

4) The obuffer contains only the first solution to the database query in order that Prolog might proceed with that solution. However, the database system must continue, in parallel, searching for the other solutions (if they exist) and must store them (in a stack) because Prolog may need them on backtracking. We will come back to this subject when we describe the database query backtracking.

5) When a query contains void variables the database system doesn't have to retrieve their values. Therefore the obuffer just has to contain the information about the result of the operation. Note that queries with only void variables can never be backtracked, so the database system must send always the <last solution> with value 1 when all the variables in the query are void.

3.4 Createrelation (Add types of arguments) Indexes in what arguments (Keys)

This question adds the specification of a database relation to the schema of the database.

In fact, the Prolog system can only send to the Scheduler the characteristics of the relation that are important in the Prolog context. These characteristics are the name of the relation, the number of fields that exist in the relation and if there exists or not the possibility of having duplicate records in the relaton table, ie, if the "insert" operation of a record that already exists in the database must be sucessful or not.

characteristics like the field names, the field types, the primary keys, the indexes, etc, must be settled by default.

The syntax of this question is the following:

<code> = 4;
<arity of the code> = 1; and
the only existing <iblock> contains the characteristics of the relation, more specifically, it will have three <smallblock>'s that will have, in this order, the name of the relation, its number of fields and the permission or not of duplicate records in its table.

Let's now see the syntax of this question by means of an example.

Example 3.4.1:

Suppose that we want to create a relation with name "person/2", with two fields and without the possibility of duplicate records. The corresponding ibuffer would be the following:

```
------------------------------------------
|  4  |  code
|-----+-----------------------
|  1  |  arity of the code
|-----+-----------------------
|  3  |  arity of the iblock
|-----+--------------------
|  9  |  length
|-----+--------
|  1  |  type
|-----+--------
|  p  |
|  e  |
|  r  |
|  s  |
|  o  |
|  n  |
|  /  |
|  2  |
|-----+--------------------
|  2  |  length
|-----+--------
|  2  |  type
|-----+--------
|  2  |
|-----+--------------------
|  2  |  length
|-----+--------
|  2  |  type
|-----+--------
|  0  |
------------------------------------------
```

Assuming that the operation was sucessful and that its relative identification number was 1, the obuffer that the Scheduler would sent in response would be:

```
------------------------------------------
|  0  |  result
|-----+--------------------------
|  1  |  relative identification
```

```
|-----+--------------------------
|  1  | last solution
|-----+--------------------------
|  0  | number of oblocks
----------------------------------------
```

Notes about the createrelation operation:

1) The effect of this operation is to create, in the data ditionary, a definition of a relation with the specified characteristics.

2) This operation is unsucessful when there already exists a definition for that relation in the data ditionary or when an internal error occurs. In these cases the error code is transmited to the Prolog system in <result>.

3) Remark that this operation changes the data ditionary by adding new definitions to it. We can only insert information in the table of a relation after creating its entry in the data ditionary.

3.5 Insert

This is one of the simplest database queries because it has no variables. Its syntax is the following:

<code> = 5;
<arity of the code> = 2;
The first <iblock> specifies the relation name that is envolved in the query; and
The second <iblock> contains the fields of the record that will be inserted in the table of that relation name.

Note that this primitive will only add records to previous existing tables. The role of creating new relations belongs to the primitive "createrelation" (described in last section) and not to "insert".

Example 3.5.1:

Suppose that, in the context of the previous examples, we want to add to the database relation "address/2" the Prolog unit ground clause: "address(Pol,Brussels)." which can be read as: "the address of Pol is Brussels".

The ibuffer that the Prolog system would sent to the Scheduler would be:

```
----------------------------------------
|  5  |  code
|-----+------------------------
|  2  |  arity of the code
|-----+------------------------
|  1  |  arity of the iblock
|-----+---------
| 10  |  length
|-----+---------
|  1  |  type
|-----+---------
|  a  |
|  d  |
|  d  |
|  r  |
|  e  |
|  s  |
|  s  |
|  /  |
|  2  |
|-----+------------------------
|  2  |  arity of the iblock
|-----+---------------
|  4  |  length
|-----+---------
|  1  |  type
|-----+---------
|  p  |
|  o  |
|  l  |
|-----+---------------
|  9  |  length
|-----+---------
|  1  |  type
|-----+---------
|  B  |
|  r  |
|  u  |
|  s  |
|  s  |
|  e  |
|  l  |
|  s  |
----------------------------------------
```

Assuming that the operation was successful and that its relative identification number was 1, the obuffer that Prolog would get as answer would be:

```
----------------------------------------
|  0  |  result
|-----+-----------------------
```

```
|  1  |  relative identification
|-----+-------------------------
|  1  |  last solution
|-----+-------------------------
|  0  |  number of oblocks
----------------------------------------
```

Notes about the insert operation:

1) The contents of the database is changed if the insertion is sucessful. Depending on the definition of the relation this operation is successful or not if the record already exists in the database. If an internal error occurs then its code will be the value of <result>.

2) In the ibuffer of the insertion there are never variables because in Prolog the external database can only contain unit ground clauses. So the obuffer has always 4 bytes: the <result>, the <relative identification> number, the <last solution> with value 1 and 0 for the <number of oblocks>.

3) The first <iblock> of the ibuffer will always consist of only one <smallblock>, that is, there are no joins for the insert operation.

4) The arguments of the relations are always atoms, ie, there can be no functors with arity greater than 0 inside those arguments.

3.6 Delete

This operation is used to delete a record in a database table. When it is called the record may or may not be completely specified. In the later case the values of the fields that were not specified must be returned to the Prolog system. The syntax of this operation is the following:

<code> = 6;
<arity of the code> = 2;
the first <iblock> specifies the relation name that is involved in the call; and
the second <iblock> contains the fields that will be deleted in the table of that relation name.

Note that as far as Prolog is concerned it makes no difference if we are deleting the last record of the relation, and it will be empty from then on, or if we are deleting one of its records and there will remain more others.

In fact, when the last record of a table is deleted, the table remains existing. The role of deleting a table is performed by "deleterelation" as we will see in the next paragraph.

Example 3.6.1:

Suppose that, in the context of the previous examples, we want to delete the information about the address of Raf but we don't know what it is.

The ibuffer that would be sent to the Scheduler would be:

```
----------------------------------------
|  6  |  code
|-----+----------------------
|  2  |  arity of the code
|-----+----------------------
|  1  |  arity of the iblock
|-----+-------------------
| 10  |  length
|-----+---------
|  1  |  type
|-----+---------
|  a  |
|  d  |
|  d  |
|  r  |
|  e  |
|  s  |
|  s  |
|  /  |
|  2  |
|-----+----------------------
|  2  |  arity of the iblock
|-----+-------------------
|  4  |  length
|-----+--------
|  1  |  type
|-----+--------
|  R  |
|  a  |
|  f  |
|-----+-------------------
|  2  |  length
|-----+--------
|  3  |  type
|-----+--------
| -1  |
----------------------------------------
```

Assuming that the operation was sucessful and that its relative identification number was 1, the obuffer that Prolog would get as answer would be:

```
-----------------------------------------
|  0  | result
|-----+-------------------------
|  1  | relative identification
|-----+-------------------------
|  1  | last solution
|-----+-------------------------
|  1  | number of oblocks
|-----+-------------------------
|  8  | length
|-----+----------------
| -1  | variable number
|-----+----------------
|  1  | type
|-----+----------------
|  L  |
|  e  |
|  u  |
|  v  |
|  e  |
|  n  |
-----------------------------------------
```

If the Prolog system was not interested in the value of variable -1, that means, if the Prolog query was: delete the address of Raf whatever it is, then the ibuffer for this query would be very similar to the one presented above with the only exception that the last <smallblock> would be:

```
----------------------
|  1  | length
|-----+--------
|  4  | type
----------------------
```

The reason for this is that the variable is now void so it doesn't matter what value it has. The obuffer, in this case, would be much more simple:

```
---------------------------------------
|  0  | result
|-----+------------------------
|  1  | relative identification
|-----+------------------------
|  1  | last solution
|-----+------------------------
|  0  | number of oblocks
---------------------------------------
```

Notes about the deletion operation:

1) The contents of the database is changed if the deletion is sucessful. This operation is not sucessful if the record doesn't exist in the database. If an internal error occurs its code will be sent in <result>.

2) In the obuffer of this operation there may exist values for variables depending on the ibuffer transmitted before.

3) The first <iblock> of the ibuffer will always contain only one <smallblock> because for this operation there are no joins of database calls.

4) Remark the difference between the two obuffers of the previous example and the effect of void variables in the obuffer of this operation.

5) Remark also that the <last solution> in the obuffer of this question is always 1 because the database system doesn't proceed in parallel with the Prolog system. It just deletes the record and stops.

3.7 Deleterelation

This operation is used to delete completely a database relation. In the answer for this query only the result is important. When this query is performed Prolog has the warranty that it can be done in order to prevent user's mistakes. The syntax of this question is the following:

<code> = 7;
<arity of the code> = 1; and
the only existing <iblock> specifies the name of the relation whose table and entry in the data ditionary are going to be deleted.

Example 3.7.1:

Let's suppose that we have the previous database with the two tables for the relations "AGE/2" and "ADDRESS/2" and that we want to delete the first one. The ibuffer for this question would be the following:

```
--------------------------------------
|  7  |  code
|-----+--------------------
|  1  |  arity of the code
|-----+--------------------
|  1  |  arity of the iblock
|-----+--------------------
|  6  |  length
```

1	type
a	
g	
e	
/	
2	

Assuming that the operation was sucessful and that it had relative identification number 1, the answer would be the following obuffer:

0	result
1	relative identification
1	last solution
0	number of blocks

Notes about the deleterelation operation:

1) The contents of the database is deeply changed if the deleterelation is sucessful. This operation is not sucessful if the relation doesn't exist in the database or if the user has unsufficient priviledge (in Unix sense) for the operation. If an error occurs its code will be sent in <result>.

2) The obuffer for this question is very short and, in fact, the only important part of it is the <result>.

3) Remark the difference between "delete" and "deleterelation". In the former case just a record is deleted and in the later case the whole relation is deleted (both the table and the entry in the data ditionary). Even when we delete the last record of a certain relation with "delete", the table continues to exist, ie, the relation is not deleted from the database schema. However, when we use "deleterelation" the table and the name of the relation no longer exist, ie, the relation is deleted from the database schema.

3.8 Backtracking

As we have seen in the retrieve operation, the obuffer of that operation only contains the first solution. However, as there may be more solutions, the database system will continue in parallel searching for them. As they are found they are stored in a stack and when Prolog asks for more solutions in backtracking they must be given back to it.

This primitive "backtracking" is the one that asks for more solutions of a previous "retrieve", therefore it has to send to the Scheduler the relative identification number of the "retrieve" that is going to be backtracked.

The syntax for this operation is the following:

<code> = 8;
<arity of the code> = 1; and
the only existing <iblock> specifies the relative identification number of the operation to be backtracked (ie, it specifies in what stack the solution is to be found).

Let's see how it works through an example.

Example 3.8.1: (backtracking without void variables)

Let's now suppose that the relation "address/2" in the database has the following table:

ADDRESS/2

name	place
Raf	Leuven
Jose	Heverlee
Yves	Heverlee

Let's suppose also that a query to the database has been made asking for the people who live in Heverlee, in Prolog syntax: "?-address(_X,Heverlee)." .

The ibuffer of this retrieve would be:

8	code
2	arity of the code

```
|  1  | arity of the iblock
|-----+---------
| 10  | length
|-----+---------
|  1  | type
|-----+---------
|  a  |
|  d  |
|  d  |
|  r  |
|  e  |
|  s  |
|  s  |
|  /  |
|  2  |
|-----+----------------------
|  2  | arity of the iblock
|-----+--------------------
|  2  | length
|-----+--------
|  3  | type
|-----+--------
| -1  |
|-----+--------------------
|  9  | length
|-----+--------
|  1  | type
|-----+--------
|  H  |
|  e  |
|  v  |
|  e  |
|  r  |
|  l  |
|  e  |
|  e  |
----------------------------------------
```

The answer that Prolog would get, assuming that this retrieve operation had relative identification number 2, would be the following obuffer:

```
----------------------------------------
|  0  | result
|-----+-------------------------
|  2  | relative identification
|-----+-------------------------
|  0  | last solution
|-----+-------------------------
|  1  | number of oblocks
|-----+-------------------------
|  5  | length
```

```
|-----+-----------------
| -1  |  variable number
|-----+-----------------
|  1  |  type
|-----+-----------------
|  J  |
|  o  |
|  s  |
|  e  |
----------------------------------------
```

After this obuffer is sent to Prolog the database system proceeds and finds the other solution and stores it in a stack.

In the meantime the Prolog system continues to run with the first solution and let's now suppose that it has the necessity of backtracking that query. So it needs another solution of the previous retrieve. The ibuffer that Prolog would send to the Scheduler would be:

```
----------------------------------------
|  8  |  code
|-----+------------------------
|  1  |  arity of the code
|-----+------------------------
|  1  |  arity of the iblock
|-----+---------------------
|  2  |  length
|-----+--------
|  2  |  type
|-----+--------
|  2  |
----------------------------------------
```

When it receives this message, the Scheduler has to see if it has any solution left in the stack of the relative identification number 2. Let us now suppose that the Scheduler has already the solution with variable -1 instantiated with "Yves" and that it knows that it is the last solution, therefore the answer to this query is the following obuffer:

```
----------------------------------------
|  0  |  result
|-----+-------------------------
|  2  |  relative identification
|-----+-------------------------
|  1  |  last solution
|-----+-------------------------
|  1  |  number of oblocks
```

```
|-----+------------------------
|  6  |  length
|-----+-----------------
| -1  |  variable number
|-----+-----------------
|  1  |  type
|-----+-----------------
|  Y  |
|  v  |
|  e  |
|  s  |
--------------------------------------
```

Let's now suppose that, in the later case, the Scheduler doesn't know yet that "Yves" is the last solution of the query then in the obuffer the position <last solution> would be 0, and let's suppose also the Prolog system backtracks again to the retrieve query. The following ibuffer would be sent again to the Scheduler:

```
--------------------------------------
|  8  |  code
|-----+-------------------------
|  1  |  arity of the code
|-----+-------------------------
|  1  |  arity of the iblock
|-----+--------------------
|  2  |  length
|-----+--------------------
|  2  |  type
|-----+--------------------
|  2  |
--------------------------------------
```

Now, the Scheduler finds that the stack of the relative identification number 2 is empty, therefore there are no other solutions. So the following obuffer would be sent to the Prolog system:

```
--------------------------------------
| -1  |  result
|-----+------------------------
|  2  |  relative identification
|-----+------------------------
|  1  |  last solution
|-----+------------------------
|  0  |  number of oblocks
--------------------------------------
```

Example 3.8.2: (backtracking with void variables envolved)

When there are void variables envolved the obuffers that the Scheduler sends to the Prolog system are shorter.

If, in the previous example the variable envolved was void, then the ibuffer of the initial retrieve would have the following <smallblock> to represent the void variable, in the place of the one where variable X is specified:

1	length
4	type

The obuffer would then be the following:

0	result
2	relative identification
1	last solution
0	number of oblocks

When there are only void variables there are never backtracking, so the former obuffer would be sent in response to the initial retrieve and Prolog would **never** backtrack that relative identification.

Notes about the backtracking operation:

1) The only backtrackable operation is the retrieve so the relative identification that will be sent to the Scheduler must correpond to a previous retrieve. If it doesn't the response of the Scheduler must have result unsucessful.

2) The Scheduler must keep the variable numbers of the retrieve operation because it has to sent them back in the backtracking obuffer.

3) The backtracking operation doesn't imply necesseraly an access to the database. The Scheduler just has to consult the stack that corresponds to the relative identification he received.

3.9 Cut

This is the simplest database query. It means that Prolog will never use again the stack of a certain relative identification so the Scheduler may discard it. Its syntax is the following:

<code> = 9;
<arity of the code> = 1; and
the only existing <iblock> specifies the relative identification of the operation whose stack is no longer needed.

Example 3.9.1:

In the context of the two previous examples let us now suppose that after the first retrieve the Prolog system sends to the Scheduler the following ibuffer:

9	code
1	arity of the code
1	arity of the iblock
2	length
2	type
2	

After receiving this buffer the Scheduler just has to discard the stack corresponding to the relative identification number 2 and, if necessary, to stop its corresponding search. The obuffer that Prolog will get is:

0	result
2	relative identification
1	last solution
0	number of oblocks

Notes about the cut operation:

1) The only possibility of unsucessful operation in what concerns the cut is a wrong relative identification, ie, a relative identification that doesn't correspond to a previous retrieve operation.

2) The cut just like the backtracking operation has only effect on relative identifications of previous "retrieves".

3) This operation doesn't need a database access. The Scheduler just has to discard the stack of the specified relative identification and, eventually, to stop its corresponding search.

4. Bibliography

<Ven84> Venken, R. and Adler, H. D., Report DB1 and DB2: the interaction between Prolog and a relational database system,
internal LOKI report (ESPRIT pp 107), feb 1984.