A PROLOG IMPLEMENTATION OF A LARGE SYSTEM ON A SMALL MACHINE

por

Luis Moniz Pereira & António Porto

UNL - 6/82 Março 82

A Prolog implementation of a large system on a small machine

Luís Moniz Pereira & António Porto

Departamento de Informática
Universidade Nova de Lisboa
Quinta da Torre
2825 Monte da Caparica
Portugal

Abstract

This paper describes a natural-language question-answering system for aiding in the planning of research investment which was completely written in Prolog (database included) and runs on a microcomputer.

We emphasize the techniques employed to get such a system to run on a small machine.

Introduction

The system we describe here is a natural language (Portuguese) question-answering system for aiding in the planning of research investment in Portugal. It knows about interactions between scientific disciplines and development goals, interactions among sciences themselves and interactions among goals. The data was gathered on several nationwide meetings, following a procedure recommended by UNESCO [1][5].

This system was implemented using a slightly modified version of Edinburgh's RT-11 Prolog (see [2]) on a small machine with a LSI-11/03 processor, 64K bytes of central memory and dual single-density floppy disks. This hardware restriction strongly influenced the design considerations, but a working system was nevertheless built which is reasonably performant.

The system is made up of two main modules: the **natural language interpreter** and the **query evaluator.**

The natural language interpreter divides itself in two sub-modules: a **lexical parser** and a joint **syntactic/semantic parser**. The lexical parser accepts input from a terminal and produces a list of morphological tokens, which are used by the syntactic/semantic sub-module to produce a Prolog goal expression which corresponds to the semantics of the natural language query.

The query evaluator includes the procedures needed to compute the Prolog goal expression coming out of the natural language processor. Any subgoal involving disk access is executed in two steps: first there is a **planning** stage whereby a new subgoal is produced, and then this new subgoal is executed — it will only access the relevant files, and timing of its operations has been optimized.

§1 The domain

The system deals with **sciences** and **development goals.**

Sciences are divided into main branches (eg. *applied sciences*), each one of these into groups (eg. *physics*) and each group into individual disciplines (eg. *optics*).

Development goals are divided into groups (eg. *agriculture*), which in turn are divided into individual goals (eg. *cereals*).

There are 110 scientific disciplines and 78 individual development goals.

The system is supposed to know about three types of correlations: between sciences and goals, between sciences and other sciences, and between goals and other goals.[1] Any hierarchical category may be referred, eg. the import of physics on agriculture, or the import of optics on development. Imports are expressed as percentage values.

§2 The natural language interpreter

The system's linguistic competence is obtained by means of a lexical and syntactic/semantic analysis, transforming a natural language sentence into a Prolog goal expression.

The modules which perform this task were adapted from a general grammar of Portuguese initially developed for another application. We shall not go here into too much detail; for that the reader is refered to [3].

[1] Correlations are normally expressed as *dependency of A on B* or *import of B on A* (which represent the same thing).

2.1 Lexical analysis

Our lexical analysis replaces words in the input sentence by their corresponding lexical categories (noun, verb, preposition, ...) with syntactic and semantic readings. This is done by making each word access a **dictionary**. If a word is not in the dictionary it is reported to the user as unknown.

Since RT-11 Prolog only gives us indexing on the predicate names, the dictionary consists of a set of single clauses for predicates whose names are the actual words we want to access the dictionary.

To each accepted word, then, is associated a dictionary clause containing all the information we will need concerning that word — lexical category, deep and surface morphology, gender, number, etc.

In general to each word corresponds a lexical entity, but sometimes several words are grouped to form a single entity, like *applied sciences* which will constitute a single noun. Then, besides having the mentioned one-word clauses we have other clauses for a single predicate which will check if a given word is followed by others that together will form a multi-word name. We could have transformed the one-word clauses to do this checking, but two extra arguments are needed — the list containing the rest of the input string and the returned sublist — which used in all dictionary clauses would take too much space given the size of the dictionary. (The **whole** dictionary must be accessible in memory for use.)

The system's vocabulary may be divided in two parts: a **core** vocabulary and a **specific** one.

The core vocabulary is independent of the system's particular domain, and divides into **linguistic** and **metalinguistic** parts. The linguistic part contains determiners, prepositions, their contractions, common verbs, relative and interrogative pronouns, phrase and sentence connectors and prelocutory expressions. The metalinguistic vocabulary contains words used to inquire the system about its linguistic capabilities.

The specific vocabulary contains all the names of the scientific disciplines and development goals, as well as nouns, verbs and adjectives used to express their relationships.

2.2 Syntactic/semantic analysis

The syntactic/semantic analysis is realized by means of a core grammar containing context-free and sensitive rules (through the **definite clause grammar** formalism) with syntactic and semantic controls. These rules handle the fundamental structures of Portuguese, in particular:

- yes-no questions;
- wh- questions;
- commands;
- affirmative, negative, relative, prepositional, coordinate, extraposed and elliptic clauses;
- complex noun complementation and adjunction;
- universal, existencial, numeral, definite and indefinite determiners;
- common verbs;
- nouns and verbs refering to the linguistic terminology.
  (allowing to ask questions about the system's linguistic competence.)

We have devoted a large energy to incorporate **elliptic** and **extraposed** structures because they are essential to a natural language interaction.

This core grammar is independent of its application and transportable to any domain, where it is completed by a specific grammar containing structures (noun and verb phrases essencially) and semantic controls refering to the particular domain.

Syntactic controls verify number and gender agreements, pointing out any faults to the user.

The syntactic and semantic analysis are not separated but blended. This solution is best to stop the parsing short as soon as a semantic error is detected, as is the case, for example, when a wrong complement for a verb is encountered; in that case, the system always informs the user what the trouble is. It also provides for a more compact grammar, which is important in this implementation

§3 The database

The system database has two distinctive parts. One contains information on the hierarchical divisions of sciences and development goals, and the other one contains the information on correlations.

3.1 Hierarchical description

Each hierarchical definition is a clause of the form

$$<code>(<number>,[<name1>,<name2>,...]).$$

$<code>$ is the internal code of some science or development group, $<number>$ is the number of elements in that group, and $<name1>$, $<name2>$, ... are their names.

$<code>$ is used as a predicate name instead of as an argument of a general predicate for efficiency reasons, already explained in the lexical analysis discussion.

The choice of internal codes was the following: `s` is the code for *sciences*, `s2` is the code for the 2nd main branch of sciences (*applied sciences*), and `s21` is the code for the 1st group within applied sciences (*physics*); `d` is the code for *development* and `d1` is the code for the 1st group of development goals (*agriculture*).

A hierarchical description clause as the one above serves two purposes: it can be used to get, through $<number>$, the codes of the elements of the group, or it can be used in the output phase to get names from codes.

The file containing the hierarchical description is consulted into central memory just prior to the query evaluation, and remains there until an answer has been produced.

3.2 Correlations

Handling of information about correlations was critical in terms of system feasability and performance, involving both space and time considerations.

The basic information available consisted of three arrays **SD**, **SS** and **DD**, containing respectively the imports of Scientific disciplines on individual Development goals, the imports of Scientific disciplines on other Scientific disciplines and the imports of individual Development goals on other individual Development goals. Each import value was given as an integer from the set {0,1,2,4}.

Given the size of those arrays (110×78, 110×110 and 78×78) and each of its elements having to be represented as a clause, there was no question of having at any one time all that information available in central memory. It was therefore necessary to split up the arrays into sub-arrays that could be individually consulted; the natural choice was to split them along the boundaries between groups of elements, and so it is that we have, for example, a file containing the imports of the disciplines of *physics* on the individual goals of *agriculture*.

In order to consult such a file when needed, its name should be related to the names of the two groups whose elements' correlations the file contains. In fact, we chose to use the name obtained by just appending the codes of those two groups — `s21d1` is the name of the *physics/agriculture* file.

What exactly does such a file contain?

First there is a clause defining the dimensions of the sub-array:

```
dim(s21d1,7,9).
```

Then there is a clause relating a general correlation predicate `cor` to a particular correlation predicate used only in the file (having the same name as the file):

```
cor(s21d1,X,Y,N/4) :- s21d1(X,Y,N).
```

When this clause is actually used `X` is bound to an integer representing one element of `s21`, and `Y` is likewise bound to an integer representing one element of `d1`. Notice that the obtained correlation value is in the form `N/4`, where `N` will be an integer in the set {0,1,2,4}; in fact all correlation values will be represented internally by a term `N/D`, although `D` may not always be 4, and only on output will such a term be converted to a percentage.

Next in the file come `s21d1` clauses for particular *non-zero* correlation values, like

```
s21d1(4,7,2) :- !.
```

The *cut* is needed because of the last clause of the file, which is

```
s21d1(_,_,0) :- !.
```

Here the *cut* is not necessary, but provides a uniform pattern for retracting `s21d1` clauses.

This arrangement allows a considerable saving of disk space, since the most common correlation value in the arrays is *zero*. The main advantage of linking `cor` to `s21d1`, instead of just using `cor`, is that clauses for `cor` have four arguments instead of three, and so we are further gaining space both on disk and in central memory when the file is consulted.

This technique of splitting the arrays solves the space problem for correlations, but not the time problem. To see why, let us have a look at how correlations are computed.

The correlation between two basic elements (eg. *optics* and *cereals*) is found by a simple lookup of a clause after consulting the appropriate file.

To find the correlation between a basic element and a group (eg. *optics* and *agriculture*) one must consult the corresponding file, get the correlations between the basic element and each element of the group, and then compute their mean value. This involves a call to the predicate `all`, which computes all solutions satisfying some goal [4].

Now to find the correlation between two groups (eg. *physics* and *sciences*), having consulted the relevant file one would have to find, for each element of one group, the correlation between that element and the other group, and then work out the final value. This involves calling `all` within `all`, which already takes some time. Worse is to find the correlation between a basic element and a super-group (eg. *cereals* and *applied sciences*), because inside the outer `all` we have, before the inner `all`, to consult a different file each time. Still worse cases are easily imagined.

The way out is to do some pre-processing of correlations once and for all, and keep those results in some files which the system knows how to access.

Having every single correlation pre-computed was absolutely out of the question, so one had to decide which correlations to pre-compute. Fortunately the regular hierarchical nature of the domain lends itself to neat and efficient solution.

Let us say that an element's level is 0 if it is a basic element, 1 if a group of basic elements, etc. Then the solution is to have a file for every two groups whose level difference is even. This guarantees that for any simple correlation computation only one file will be consulted and no more than one call to `all` will be executed. (Of course complex questions may involve several simple correlation computations.)

The initial files already agree with this definition, and so no file has to receive special treatment from the system. We just have to generate the remaining files from the initial files. Two special "groups" have however to be considered, one having *sciences* as its unique element and the other one having *development* as its unique element. We named them respectively `x` and `y`. As an example, the file `sy` contains the correlations between the main branches of *sciences* and the *development*.

§4 Query evaluation

The goal expression which comes out of the natural language procesor eventually contains calls to evaluate correlations. All such calls are of the form

```
correlation(X,Y,V)
```

where `X` and `Y` define what are the elements whose correlation is wanted and `V` expects its value (in the form `N/D`).

As we said before, the evaluation of `correlation` proceeds first through a **planning** stage before actually consulting the relevant files and getting the correlation value (using `cor`).

The general philosophy here is to compute first everything which is deterministic, except calls which must wait for instantiations in its parameters (eg. arithmetic calls), and then proceed to the evaluation of postponed goals, having set the optimal order of execution among them.

Three main tasks are carried out in the planning stage:

▸

Getting the names of files to be consulted — this is roughly done by looking at the names of the elements whose correlation one wants to compute: they allow us to compute the corresponding levels, the level difference shows what type of file is needed, and, accordingly, the names one has to append to get the file name are either the names of the elements or those of their groups, which are easily computed from theirs.

▸

Getting goals to generate the elements of a group — if the predicate `all` is going to be used there will be the need for this; from the hierarchical description clause for the group, or from the `dim` clause of a file involving the group, one can access the number `N` of elements in the group, this being the only information needed to construt a goal generating integers from 1 to `N`, which will represent the elements of the group within `cor`.

▸

Setting the right order of execution among goals — of the goals to be postponed the system knows which ones produce instantiations to be used by others, so it can preview the future optimal execution and set up the goal expression for doing so.

These three tasks are done concurrently. Far from using a general query planner we just wrote clauses for `correlation` which perform the planning in a highly optimized way for the tasks at hand.

§5 System performance

To get the system to run with the small amount of central memory available one had to resort to the technique of chaining various modules (which are thus separate programs), using a disk file to pass information from module to module. This was easily achieved by making RT-11's procedure for chaining programs available inside Prolog.— the goal `chain(`<*savfile*>`,`<*consultfile*>`)` launches <*savfile*> which is a Prolog program that initially consults <*consultfile*>.

Three modules are chained: the **lexical parser** (because of the size of the dictionary), the **syntactic/semantic parser** and the **query evaluator**, which chains again to the lexical parser for the next query.

The main consequence is that a great part of the time required for the system to answer a query is lost in the chaining process of loading the next module from disk.

A typical 100 character query takes about 16 seconds to be answered, thus distributed:

Lexical analysis	3
Chaining	4
Syntactic/semantic analysis	2
Chaining	4
Query evaluation (1 file consulted)	4

The system will ultimately be installed on a PDP-11/23, where extended memory, faster processor and better disk access will certainly boost the current performance. In particular, chained modules will be permanently in memory, making chaining virtually instantaneous.

Acknowledgements

This work was done under contract with the Junta Nacional de Investigação Científica e Tecnológica (JNICT).

We thank John McCarthy for providing the occasion and facilities to prepare this manuscript at Stanford University.

References

[1] Caraça, J.M.G. ; Pinheiro, J.D.R.S. — *Prioridades em ciência e tecnologia — Identificação de áreas prioritárias para I&D*

Junta Nacional de Investigação Científica e Tecnológica 1981

[2] Clocksin, W. F. ; Mellish, C. S. — *Programming in Prolog*

Springer-Verlag 1981

[3] Pereira, L.M. ; Oliveira, E. ; Sabatier, P. — *An expert system for environmental resource evaluation through natural language*

submitted to First International Conference on Logic Programming, Marseille, 1982

[4] Pereira, L. M. ; Porto, A. — *All Solutions*

Logic Programming Newsletter, n. 2, Autumn 1981

[5] UNESCO — *Méthode de détermination des priorités dans le domaine de la science et de la technologie*

Études et documents de politique scientifique n. 40, UNESCO 1977