The IDEA User Interface: the Power of Logic Programming in GUI Implementations *

Mirko Sancassani, Giovanna Dore, Ugo Manfredi
DS logics S.r.l.
Viale Silvani, 1 - 40122 Bologna (Italy)
[mirko, dore, um]@dslogics.it

July 5, 1994

Abstract

This paper is focused on the Graphic User Interface of IDEA, a consultation system designed to provide a user-friendly environment to compute and analyze statistical data, and shows how Prolog and a set of extensions to this language provided by the APPEAL environment, work together making it easy the develop of complex user interfaces. Here we refer to an application of this system in the epidemiological field.

1 Introduction

IDEA is a decision support system which allows a manager to *compute* and *analyze* statistical data, which we call *indicators*, relative to its specific domain. IDEA *analysis* sub-system concerns the formalization of methods and rules to support a correct interpretation of the computed indicators. IDEA *computing* sub-system, whose user interface description is the subject of this paper, provides the user with an easy and uniform way to compute and report indicators based on data managed by heterogeneous local and remote DBMS's.

IDEA computing system can be thought of as an *interpreter* who tries to put the application-domain world in touch with the physical database world. Using its knowledge IDEA is able to understand a user request, specified using the application-domain jargon, and to find one or more computational ways to solve a query.

IDEA includes application-domain knowledge and technical knowledge about heterogeneous data organization and access methods. Its knowledge bases are modeled

*This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 92.01662.69 and by the Health Ministry of Emilia Romagna, Dr. Barbolini.

according to the *frames* paradigm of knowledge representation [5]. A detailed description of IDEA knowledge bases can be found in [9]. For our purposes, it is enough to know that they are implemented in SICStus objects, a standard library which provides an Object-Oriented version of SICStus Prolog [2]. Physical queries to the databases are expressed in IDEA knowledge bases using *LogicSQL* [7], a database query language which allows to express complex SQL queries within a logical paradigm.

The power of IDEA can be easily and completely exploited by the final user thanks to an intelligent graphic interface, through which the user can query the system and generate reports.

The following paper describes IDEA user interface, which is implemented in APPEAL [3]. APPEAL is a programming environment built on top of SICStus Prolog, which put at programmer's disposal full access to the functionalities of the X-Window System at the toolkit level [10] by using a Prolog integrated language called Widget Description Language (WDL) . WDL provides a declarative way of specifying and manipulating widgets, associating the execution of Prolog code to relevant interaction events. For IDEA project we choose the OSF Motif toolkit [4], which has become a standard for Unix GUI applications.

IDEA User Interface can be considered an example of how Prolog's computational model is suitable to satisfy different programming requirements for the interface implementation. Specifically, we needed a programming language with graphical capabilities, parsing capabilities, suitable for knowledge representation and which could be profitably used as database query language. We found in the logic paradigm all these potentialities: Prolog augmented with extensions such as DCG, SICStus objects, WDL and LogicSQL proved to be a very powerful, flexible and uniform programming language. This made the problems posed by IDEA's User Interface design and implementation easy to solve.

IDEA User Interface consists of the *Query Editor*, the *Output Interface* and the *Visualization Package*. The user can query the system by using the Query Editor. By using the Output Interface, he can select the desired output type and data organization and he can call tools of the Visualization Package to map the query results on the screen.

Even though IDEA's interface was designed as a set of general, domain-independent tools, throughout this paper we will refer to one of the current IDEA's applications: the computation and analysis of epidemiological indicators.

Epidemiology is a science which tries to plan governmental health activities on the base of selected statistical information about population, mortality, diseases etc. Such pieces of information are called *epidemiological indicators* and are the result of a calculus that involves one or more *variables*. Example of an epidemiological indicator is the *mortality rate*.

A variable is a relevant epidemiological entity, like population or mortality, whose value can be directly computed from database data. Variable and indicator values are constrained by some parameters, like sex, age or place, called *attributes*. Furthermore, attribute values can belong to different *granularities*. An attribute granularity is the level of description of an attribute value. For example, a correct value of the attribute `place` for the granularity `nation` is `Italy`. From the computational point of view

a variable or an indicator can be seen as a function with attributes as independent variables.

The paper is organized as follows: the following section gives a description of the User Interface features we wanted to achieve in IDEA. Section 3 and 4 describe respectively IDEA Query Interface and some of its implementation issues. Section 5 describes IDEA Visualization Package, while the Output Interface is the section 6 topic. Conclusions and future works are considered in the last section.

2 IDEA User Interface Requirements

In the development of IDEA's project, we paid special attention to the user interface since a system like IDEA, which will be used in application fields where people have no information technology skills, depends dramatically on the quality of the user interface. More precisely, the user should be able to use the interface even without a training. This is possible if the interface is *self explanatory* and guides the user in his interaction with the system. In this way, inconsistent user-interface configurations are avoided.

The main IDEA's User Interface aim is to allow a complete access to the system functionalities in the easiest way. Graphic windowed systems are the best tools to achieve these goals. In particular, IDEA User Interface:

speaks the user jargon: all the output messages the user can read from the User Interface panels or enter in the system belong to the application-domain jargon. Information Technology terms are avoided. For instance, diseases are often identified with a numeric code, but the user can enter the disease description, not the code. If the user does not know correct values, they can be requested to the system.

does not bother with useless questions: IDEA User Interface solves, when possible, input or computational ambiguities by asking the internal knowledge bases, not the user.

is propositive, not inquisitive: the interface drives the user towards a query solution. Thus, each time informations are requested to the user, the interface explains what the system is doing and proposes possible alternatives among which the user can choose. For example, when IDEA can solve a query by using alternative sets of databases, the User Interface displays the possible database set names once at a time for user's approval.

is user error-proof: several interactive helping tools are provided by IDEA User Interface to reduce error probability. Anyway, if an error occurs, a detailed explanation message helps the user to rapidly understand which specification must be corrected. IDEA does a knowledge based syntactical and semantical check control of the entered specification.

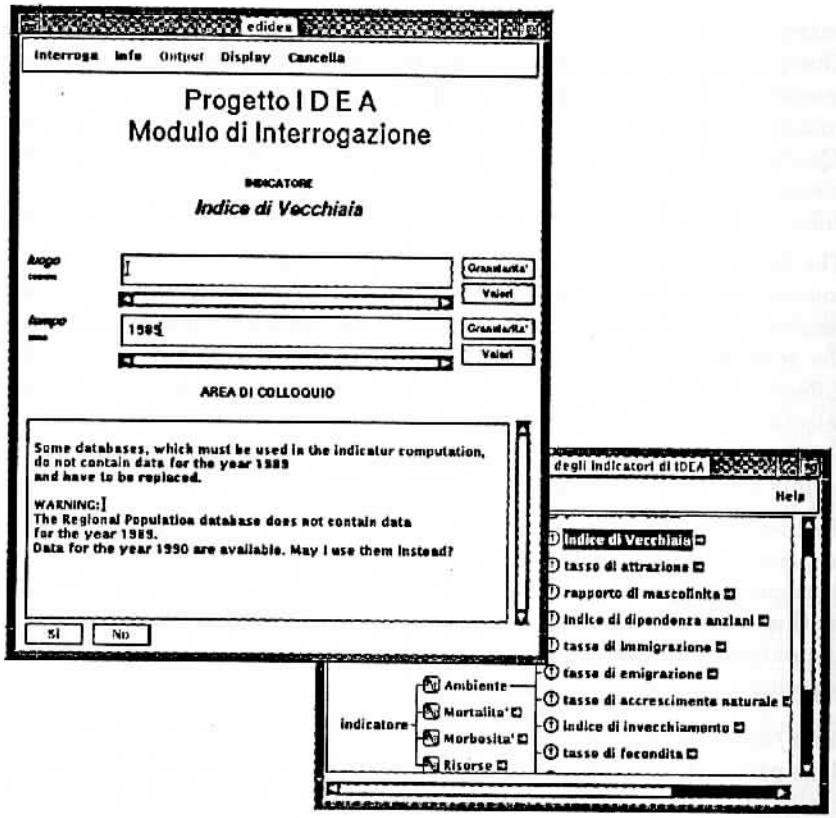


Figure 1: A Query Editor instance and an IDEA's Browser

From the technical and implementative points of view, IDEA's code of the User Interface is completely *general*, i.e. it does not depend on the knowledge domain of IDEA's application. The application domain configurability of the interface tools is achieved by using IDEA knowledge bases.

3 A description of IDEA Query Interface

By selecting an appropriate button of the IDEA main panel, an Indicator Browser (see figure 1) is mapped on the screen. The Indicator Browser offers a tree shaped representation of the computable-object names. The user can move the cursor on an object label and choose, by means of a popup menu, the query option to make the Query Editor appear (see figure 1). The Query Editor instance is built by consulting IDEA knowledge bases and, according to the information retrieved, is set to a default configuration. It looks like a partially filled input window which provides the user with indicator related information. It is made of two main areas:

the entry area:

This part looks like a form which can be filled by the user. An entry field, a granularity label, a granularità (granularities) button, and a valori (values) button are associated with each indicator attribute. Selecting the Interroga (Query) option and leaving the attribute input fields unfilled, the most general computation is performed, i.e. IDEA will compute an indicator value for each different combination of attribute values available for the selected granularity.

The Input fields allow the user to specify constraints on attribute values. Such constraints can reduce the number of computed tuples or can aggregate partial results in different ways. Selecting the granularità button, the user can see all the correct granularities for an attribute in a list widget and can switch to a different granularity with respect to the one shown in the granularity label. The selection of the valori button makes visible, in a list widget, all the possible values of an attribute granularity. The user can put values in the entry field by simply selecting values with the mouse and clicking the appropriate button.

Furthermore, the user can specify complex attribute constraint configurations by using a simple syntax. For example, he can write `Bologna, Modena, Bologna+Modena` in the `place` entry field and `1990` in the `time` entry field. In this case, the indicator computation will produce three tuples representing the oldness index in the year 1990 for Bologna, Modena, Bologna and Modena together.

the user/system interaction part:

This part, activated when the user runs an indicator computation, manages the conversation between the user and the system. It is made of an output text area where the system reports error messages, correction hints, computational information, solution proposals, etc., and of a couple of buttons by selecting one of them the user can accept or refuse the system proposals.

4 Implementation Issues of IDEA Query Interface

Using the functionalities of the *Widget Description Language* it was very easy to realize the graphic object (widgets) structures of the Query Interface. In WDL, a hierarchy of widgets can be described as a single Prolog term for the predicate `widget/2`, called *Widget Structure Description Term* (WSDT). To show the expressiveness and compactness of WDL, let's us consider the following code fragment implementing the Query Editor's main widget:

```
shell widget InsPoint=queryWidget(Id, IndName) :-
  Id = applicationShell/[input(true), width(500), ...]
  - [ xmMainWindow
      - [ xmRowColumn/[rowColumnType(menu_bar)]
          - [ xmCascadeButton/[labelString('Interroga'),
                               activateCallback(g(query(TextId, IndName)))],
              xmCascadeButton/[labelString('Output'), ...] ...
```



```
            ],
          InsPoint=xmForm,
          xmRowColumn
          - [ TextId = xmText/[heigh(100), ...],
              xmPushButton/[labelString('Si'), ...],
              xmPushButton/[labelString('No'), ...]
            ]
      ]
  ].
```

The first argument of the predicate `widget/2` is the atom shell and it means that the widget is a top-level window widget. The second argument allows the definition of the widget name (`queryWidget`) and of input and output parameters. In our case, `queryWidget` is the name of the widget, `IndName` is an input parameter while `Id` is an output variable returning the widget identifier. Moreover, the second argument defines the widget which is the *Insertion Point* of this window structure, in our case the *form* widget whose identifier is `InsPoint`. The insertion point can be used to dynamically insert new widgets inside the form.

The body of the WSDT describes the size and composition of the Query Editor window which contains a Motif **XmMainWindow**. Two **XmRowColumn** and a **XmForm** widgets are defined as **XmMainWindow**'s children. The first **XmRowColumn** contains the *button* widget implementing the top menu of the Query Editor. To each button is associated a Prolog goal which will be executed when the button is selected. The second **XmRowColumn** contains the message area and the **'Si'**, **'No'** buttons.

Widget programming techniques in WDL were presented in [8] and they are not subjects of this paper. However, it is worth to point out how the WDL *Insertion point* feature is used to customize the Query Editor at run time on the basis of the information stored in the IDEA knowledge bases.

The Query Editor form is defined by two widgets: the general shell widget we described in the above example and a widget defining all the graphic objects related to an attribute description. When the Query Editor for an indicator or variable is requested, the predicate `edidea(<Indicator Name>)` is called. `edidea/1` is implemented as follows:

```
edidea(IndName) :-
   shell widget InsPoint=queryWidget(Id, IndName), % WDL call
   assert(IndName, Id), % Prolog
   IndName::attribute(AttrName, GranMin), % SICStus objects call
   InsPoint widget attributeWidget(AId, AttrName, GranMin), % WDL call
   fail.

edidea(IndName) :-
  retract(IndName, Id), % Prolog
  Id wproc map. % APPEAL graphic procedure call
```

The first call to `widget/2` realizes the Query Editor shell widget and gets the Insertion Point widget identifier. Then, the indicator Knowledge Base is asked for

an indicator attribute name by sending the message `attribute/2` to the SICStus object `IndName`. For each attribute, the call to `InsPoint widget attributeWidget/3` realizes the opportune set of widgets inside the widget `InsPoint`. When no more attributes for `IndName` are found, the second `edidea/1` clause is executed and the whole Query Editor window is mapped on the screen by calling the APPEAL graphic procedure `map`.

The use of WDL programming techniques allows us to keep the Query Editor code independent from IDEA's specific application domain.

The above code points out how the communication between the graphical interface and the IDEA knowledge bases is performed. In fact, IDEA knowledge bases are implemented in SICStus objects, while we prefered to encapsulate the whole user interface code in a single SICStus module. We did not implement the User Interface in SICStus objects because the syntax burden and the lack of an effective debugging tool would not be repaid by actual advantages. Anyhow, the flexibility of the SICStus Prolog language allows us to send, when needed, messages to SICStus objects from the User Interface module by using the SICStus objects operator "::" for sending messages.

An example of the computational power of mixing Prolog, WDL, SICStus objects, DCG and database commands within the Prolog computational model is represented by the following code which implements the query algorithm:

```
query(QEId, IndName) :-
  read_constraints(QEId, Constraints),
  parse_constraints(Constraints, Query),
  syntactic_check(Query),
  semantic_check(Query),
  assertz(query(QEId, Query)),
  fail.

query(QEId, IndName) :-
  retract(query(QEId, Query)),
  IndName :: Query,
  put_in_tmp_file(Query),
  fail.

query(QEId, IndName) :-
 copy_to_ingres_rel(QEId),
 !.
```

When the user selects the *Interroga* button of the Query Editor, the `query/2` predicate is called. `QEId` is the identifier of the Query Editor text widget used by IDEA to communicate with the user and it is also taken as query identifier. `read_constraints/2` uses APPEAL graphics primitives to read the attribute constraints from the attribute entry fields. `parse_constraints/1` calls a parser written in DCG which returns in backtracking a set of IDEA's queries, since a query expressed by using the Query Editor can correspond to several queries to the IDEA knowledge bases. `syntactic_check/1` asks the IDEA knowledge bases for the correctness of

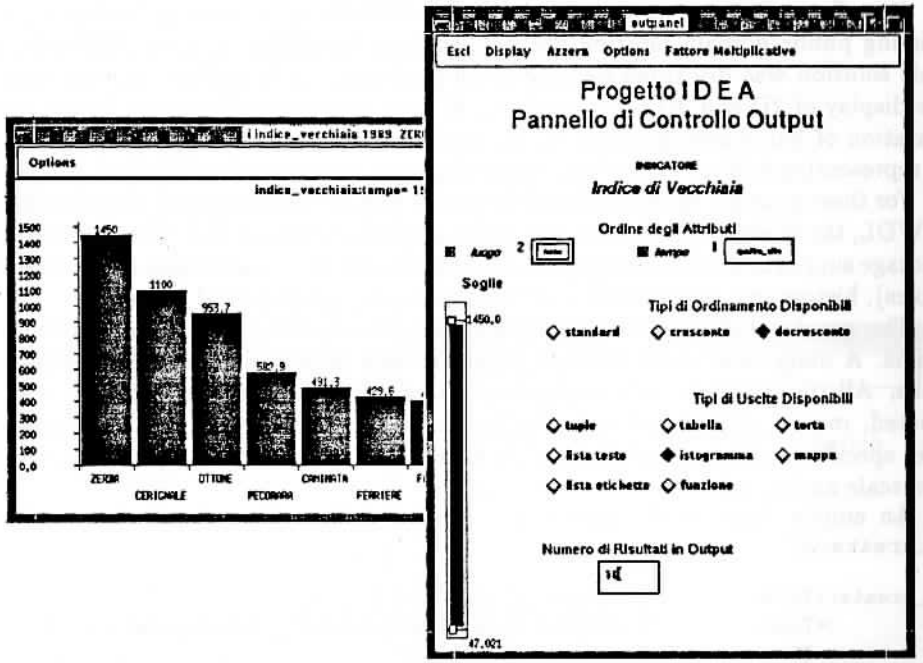


Figure 2: An Histogram instance and the Output Configuration Panel

attribute values. If any value is not recognized by the system, `syntactic_check/1` fails sending messages to the Editor Message area. `semantic_check/1` asks the IDEA knowledge bases for existing databases which can satisfy the query constraints, if this is not the case, alternative queries are proposed in backtracking to the user. `assertz(query/2)` asserts the query in the internal Prolog database. The query is not immediately computed since we prefer to complete the whole input data checking in order to avoid partial computations. By backtracking all IDEA's queries are checked and asserted.

The second `query/2` clause simply sends the query to the correct SICStus object for the computation. The resulting tuples are obtained by backtracking and are stored in a temporary file.

The third `query/2` clause copies the resulting tuples in an INGRES relation by using predicates of the Prolog/Ingres [1] interface provided by APPEAL.

5 The IDEA Visualization Package

The **Visualization Package** was developed to support in an easy way the visualization of indicators and variables in tabular or graphical form. Though developed with the IDEA needs in mind, it is a *general purpose* package that can be used even to represent completely different sets of data.

As a first attempt, we evaluated the opportunity of interfacing Prolog with an existing public domain plotting package, such as **GnuPlot** [11] or **PLPLOT** [6]. This solution was discarded because these packages, while performing very well on the display of 2D and 3D functional or grid data, are very rudimentary in the representation of bar charts and offer no support at all for visualizing geographical maps or representing tables, while these types of output were mandatory in our project.

For these reasons, we decided to develop the graphic tools required through the use of WDL, the Widget Description Language, previously introduced. The Visualization Package supports the following types of graphical output: spreadsheets (bidimensional tables); histograms (bar charts) and functions; pies; geographical maps.

The graphical objects manipulated by the Visualization Package are called *diagrams*. A diagram is contained in a graphical shell with a menu of selectable operations. All the diagrams offer a standard set of operations or methods: they can be created, moved, resized, printed (PostScript) and destroyed. Certain diagrams also offer specific methods to configure their aspect, add graphic options, force the choice of a scale and so on.

An empty diagram of a given type is created by calling the Prolog predicate `vp_create/3`:

```
vp_create(+Title,       % diagram title (shell title)
          +Type,        % diagram type ('spreadsheet', 'histogram', ...)
          -ID)          % diagram identifier
```

There also exists a predicate `vp_create/7` which gives more control on the created diagram and allows the user to install his own menus and menu-items on the diagram.

Once created, the diagram must be filled with the data to be displayed. The predicate that does the display depends, in general, from the diagram type.

The **spreadsheets** can be directly represented with a Motif row-column widget: this works well if the number of items in the spreadsheet does not exceed one thousand or so. Bigger spreadsheets are represented in textual form.

Histograms, functions and **pies** were implemented by using the APPEAL *Picture Widget*, a widget offering a set of drawing primitives.

A **geographical map** is a very effective way to represent data that vary from place to place. This diagram can represent arbitrary geographical regions whose contours are expressed in the UTM (Universal Transverse Mercator) cylindrical coordinate system. This diagram poses important efficiency problems, because the amount of data needed to correctly represent a region can be bigger than 1MB. The Prolog I/O primitives are just too slow to manipulate such an amount of data each time. For this reason, and also to facilitate high-quality printing of the chart, we decided to directly implement this diagram in PostScript. Contours data and PostScript procedures required for their interpretation are kept in a set of (fixed) files, while only the (variable) actual calls to the procedures are produced each time by the Prolog program. The subset of data and procedures needed for each specific display can be sent directly to the printer or to a special widget able to directly interpret PostScript code (the Ghostview widget) for a direct visualization.

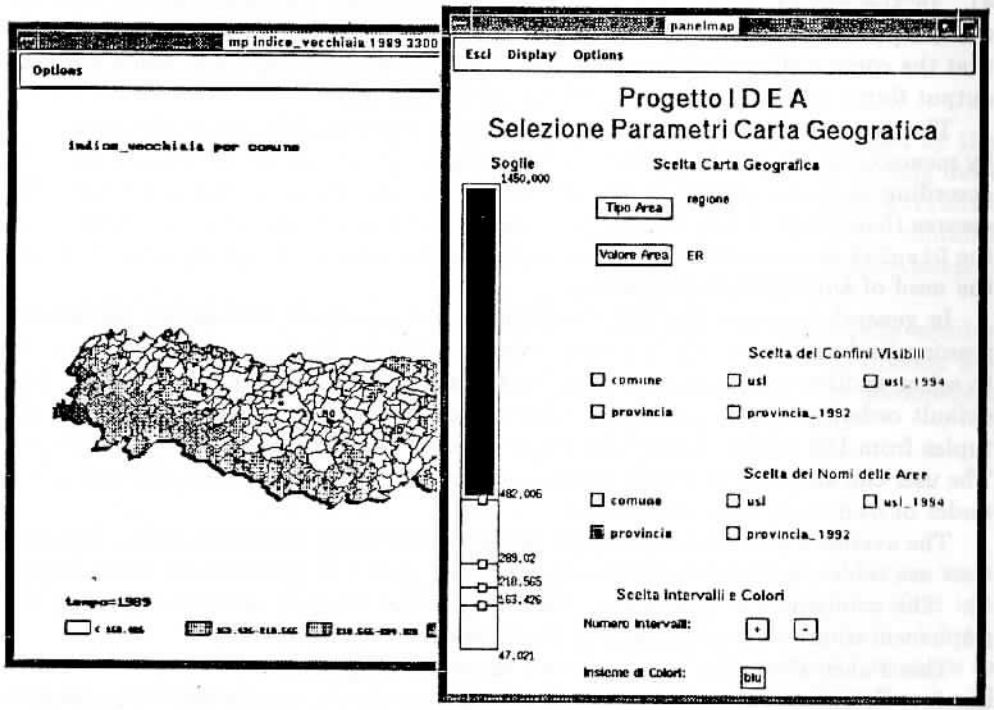


Figure 3: A Map instance and IDEA's Map Panel

6 The Output Interface

The rationale underlying the IDEA Output Interface design was to provide the user with a sufficient variety of graphic visualization tools and with the possibility of organizing and selecting output data accordingly to its wishes.

A desirable feature of the Output Interface is the ability to display and compare different data representations and organizations without having to repeat the whole computation of an indicator (variable). This means that the computational process is split into *two phases*: in the *first* phase the indicator (variable) is computed and the results produced are stored internally in a temporary Ingres relation (see section 4). In the *second* phase such results can be organized and manipulated through the Output Interface in different ways. This mode of operation has the advantages that the computationally intensive phase (the first) is done only once, while different output forms can be produced with little additional cost.

The user can select the desired output type (histogram, pie, geographic map, ...) by means of an *Output Configuration Panel* (figure 2). The panel auto-configure itself according to the Knowledge Bases and the current content of the Query Editor. This assures that a *default output* for a given query always exists: the user can always press the `Display` button obtaining a meaningful visualization of the query values without the need of any further configuration.

In general, however, the default output is too generic or undesirable for various reasons, and in this case the user may interact with the Configuration Panel in order to select another output type or organize the attribute values in a different way. The default ordering of the query attributes coincides with the extraction order of the tuples from DB tables: this can be easily changed through the Configuration Panel. The user can also sort the result values in ascending o descending order, discard values under or over a *cut value* and so on.

The available IDEA's output types are those offered by the Visualization Package, that are tables, histograms (figure 2), functions, pies and geographical maps (figure 3). The configuration parameters needed by maps are very specific, requiring the implementation of a particular Map Configuration Panel (figure 3).

This Panel allows the user to choose the main geographic area that he wishes to display. He can also select interactively the number of color ranges that will appear in the map and the numeric thresholds for those ranges: all the areas having the value in the same range will be filled with the same color. The user can also force (and remove) the display of the names of the geographical entities, of the administrative boundaries, and so on.

The output of phase 1 (computation of queries) was stored, in the first implementation of the system, in the internal Prolog database, by using the predicates *recorda/recordz*. This solution lead us into troubles, due to the size of the output, producing an unacceptable growth of the memory size allocated to the process. The solution was also unsatisfactory because most of the times the ordering and selection operation acting on stored values were necessarily done accessing the Prolog facts in sequential way.

The solution to this problem was to store the results of the queries in temporary

files. We can directly call the Ingres *copy* routine that loads the file content into a database table: this operation is very fast and the problem of memory size is radically solved. The table can be now directly manipulated with LogicSQL to extract the required columns, order them, cut unwanted values in a simple and efficient way.

The following code fragment shows how an operation of data ordering and cutting is done by the Output Interface:

```
qi_query(Table, List) :-
  recorded(Table, options(OrderAttrList, Cut1, Cut2), _),
  lsql(select List from [Table]                          % LogicSQL call
       where [Table.value >= Cut1, Table.value =< Cut2]
       orderby [OrderAttrList]),
  display_output(List).
```

First of all, `qi_query` retrieves Prolog facts storing the choices performed by the user in the Configuration Panel, then executes a LogicSQL query for extracting data from the Ingres temporary relation `Table` in the order specified by `OrderAttrList` and cutting values under `Cut1` and over `Cut2`. The output parameter `List` contains values returned by the LogicSQL query and is used by the predicate `display_output` that produces the graphical output selected by the user.

As the previous example shows, the LogisSQL language is structurally similar to SQL and allows to express SQL queries in a flexible way: query parameters are expressed by logical variables that can be used both as input or output parameters depending on whether they are instantiated or not at run-time.

7 Conclusions and Future Works

The design and implementation of IDEA and its user interface was a major task. Chosing the most suitable programming paradigm and language was therefore important. We needed a language, allowing us to easily implement graphical objects, suitable for knowledge representation, by which it was easy to implement languages to make user and system communicate, whose interaction with external DBMS was well integrated in the language computational model. Prolog extended with some software engineering tools (DCG, WDL, Objects, LogicSQL) gave us all these features and more, as the possibility to implement the project through incremental prototypes.

Although satisfied of the final outcome, we are planning some improvements and extensions of the IDEA User Interface. A knowledge base graphic editor will allow the user to define complex views involving a set of indicators whose values will be displayed in a unique diagram. We plan also to improve the Visualization Package to support more sophisticated output facilities such as multiple histograms and 3D diagrams.

8 Acknowlegments

We thank the Epidemiological Observatory of Emilia Romagna, especially Barbara Curcio Rubertini, Eleonora Verdini, Lino Falasca, E. Di Ruscio and Massimo Negri who sponsored this project and gave us many interesting suggestions.

Finally, Cristina Ruggieri gave us useful hints on the overall presentation of our work.

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 92.01662.69 and by the Health Ministry of Emilia Romagna, Dr. Barbolini.

References

[1] Ingres Corporation. *Ingres: The Intelligent Database.* Ingres Corporation, 1991.

[2] M. Carlsson and J. Widen. Sicstus Prolog user's manual. SICS research report R88007C, Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden, July 1990.

[3] DSlogics. Appeal 2.1 user's manual. CNR Technical Report of Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo R-04-092, DS logics S.r.l., Viale Silvani 1, 40123 Bologna ITALY, 1993.

[4] Open Software Foundation. *OSF/Motif Programmer's guide.* Prentice Hall, Englewood Cliffs, N.J. 07632, 1990.

[5] M. Minsky. A framework for representing knowledge. In P.H. Winston, editor, *The Psychology of Computer Vision.* McGraw Hill, 1975.

[6] G. Furnish M.J. LeBrun and T. Richardson. The plplot plotting library programmer's reference manual. Report, University of Texas at Austin, July 1993.

[7] U. Manfredi and M. Sancassani. LogicSQL : Augmenting sql with logic. Technical report of progetto finalizzato informatica e calcolo parallelo, CNR, Piazzale Aldo Moro 7 - Roma, July 1993.

[8] C. Ruggieri and M. Sancassani. A set of Prolog programming tools. In *Proceedings of the first International Conference on the Practical Application of Prolog*, April 1992.

[9] C. Ruggieri and M. Sancassani. IDEA: Intelligent data retrieval in prolog. In *Proceedings of the 1994 Joint Conference on Declarative Programming GULP-PRODE'94*, September 1994.

[10] F. Russo. A declarative x-window interface for prolog. In *Proceedings of the seventh Italian Conference on Logic Programming*, June 1992.

[11] T. Williams and C. Kelley. *GNUPLOT An Interactive Plotting Program.* Free Software Foundation, 1993.

A Sleeper-based Prolog Interpreter with Loop Checks*

Filomena Ferrucci° Vincenzo Loia° Giuliano Pacini°° Maria I. Sessa°

°° Dipartimento di Matematica Applicata ed Informatica
Università di VENEZIA - I
Via Torino, 155 - 30170 Mestre (VE) - I
pacini@di.unipi.it

° Dipartimento di Informatica ed Applicazioni
84081 Baronissi SALERNO - I
{filfer;loia;mis}@udsab.dia.unisa.it

Abstract

An approach attempting to solve the problem of non termination of a query to a logic program consists in modifying the computation mechanism by adding a capability of pruning. These mechanisms are called loop checks, as they are based on excluding some kinds of repetitions in the SLD-derivations. Loop Check mechanisms are not simple to implement since it is necessary to explore past derivation steps still active in the history of the Prolog derivation. In this paper we present an efficient implementation of the loop checks based on the equality of goals/resultants in a Prolog interpreter. A high level control mechanism, named Sleepers, is used as basic implementation tool which allows to handle in a simple and practical way the history of the computation, satisfying the needs of simplicity, extendibility and efficiency.

* This research has been partially supported by grant 40% MURST

1 Introduction

A Prolog interpreter is based on the depth-first search of the SLD-tree. This may cause the missing of a solution when the proof is focused on an infinite branch. The problem of detecting any infinite branch in an SLD-tree is obviously undecidable, as the logic programming has the full power of recursion theory. An approach attempting to solve this problem consists in modifying the computation process by adding a capability of pruning, i.e., at some point the interpreter is forced to stop its search through a certain part of the SLD-tree [2,3]. This is obtained by adding to interpreter a mechanism that is called loop check, as it is based on excluding some kinds of repetitions in the SLD-derivations. The purpose of a loop check is to reduce the search space for top-down interpreters in order to obtain a finite search-space, without loss of results of the refutation process. Thus, the completeness property of a loop check concerns with the capability of pruning every infinite SLD-derivation. In contrast, the soundness property of a loop check is concerned with the preservation of the results (successful, computed answer substitution). Some sound loop checks which are proved complete for some classes of logic programs can be found in literature [2, 3, 6].

In this paper we present how some loop checks can be embedded in a Prolog interpreter thanks to the "Sleepers", a high level control mechanism which allows to handle in a simple and practical way the history of computation [8].

The aim of this paper is to demonstrate how it is possible to implement a complete Prolog interpreter that is able to detect and manage several loop checking mechanisms, without affecting the high expressiveness and efficiency of the underlying Prolog interpreter. Indeed, all the choices at the basis of the implementation of the interpreter, e.g., the memory and control flow management, are not modified or influenced by the loop check handling.

The paper is organized as follows. First a synthesis of the loop checking technique is presented in Section 2. In Section 3 we briefly discuss our software-oriented strategy which allows to solve in a high level fashion the implementation problems which occur when a Prolog interpreter is augmented with a loop checking mechanism. Implementation details of the Prolog interpreter are provided in Section 4. Loop checks implementations are discussed in Section 5 and Section 6. An important optimization is briefly explained in Section 7. Conclusion and future works of Section 8 close the paper.

2 Loop Checks

In the paper the basic logic programming terminology is used as described in [1, 7]. In particular, the pre-ordering $\leq$ (*more general than*) on the substitutions is such that for two substitutions σ and τ, $\sigma \leq \tau$ iff there exists a substitution ρ such that $\sigma\rho = \tau$ (ε will denote the empty substitution). For two expressions E and F, we write $E \leq F$ (F is *less general* than E) when F is an instance of E, i.e., there exists a substitution ρ such that $E\rho = F$. The notation $G \Rightarrow_{C,\theta} G'$ represents an SLD-derivation step from a goal G to a goal G' using a clause C and an mgu θ. Given a logic program P and a goal G_0, $D = (G_0 \Rightarrow_{C_0,\theta_0} G_1 \Rightarrow \dots)$ is an SLD-derivation of $P \cup \{G_0\}$. As usual the goals G_0, G_1, ... are called *resolvents*. When one of the resolvents G_n is empty ($G_n = \square$) it is the last negative clause of the derivation. Such a derivation is then called an *SLD-refutation* (or a

successful SLD-derivation), the composition of mgu's $\theta_0\theta_1 \dots \theta_{n-1}$ restricted to the variables of G_0 is the *computed answer substitution*. and $|D|$ stands for the length of the refutation, i.e., the number of derivation steps.

In this section some definitions and results about loop checking are recalled, as they are presented in [2, 3]. A loop check is defined as a set of SLD-derivations that are pruned exactly at their last node. More formally, the following definitions are given:

Definition 2.1. Let L be a set of SLD-derivations. *RemSub(L)* = {D $\in$ L | L does not contain a proper initial subderivation of D}. L is *subderivation free* if L = RemSub(L). ❑

Definition 2.2. A *simple loop check* is a computable set L of finite SLD-derivations such that L is subderivation free and closed under variants. ❑

Definition 2.3. A *loop check* is a computable function L from programs to sets of SLD-derivations such that for every program P, L(P) is a simple loop check. ❑

Definition 2.4. Let L be a loop check. An SLD-derivation D of $P \cup \{G\}$ is *pruned by L* if L(P) contains a subderivation D' of D. ❑

The second condition in Definition 2.2 ensures that the choice of the names of variables in the input clauses in an SLD-derivation does not influence its pruning. Note that a simple loop check can be seen as a loop check, namely as a constant function. Thus a loop check is called simple when its behaviour does not depend on the program the interpreter is confronted with. The set of SLD-derivations L(P) in Definition 2.4 can be extended in a canonical way to a function $f_{L(P)}$ from SLD-trees to SLD-trees by pruning in an SLD-tree the nodes in {G | the SLD-derivation from the root to G is in L(P)}.

The basic properties of a loop check are soundness and completeness. The property of soundness is concerned with the preservation of the results (successful, computed answer substitution). In contrast, the completeness concerns with the pruning of every infinite SLD-derivation. More formally, we have the following definitions.

Definition 2.5.

i) A loop check L is *weakly sound* if for every program P, goal G and SLD-tree T of $P \cup \{G\}$: if T contains a successful branch, then $f_{L(P)}$ (T) contains a successful branch.

ii) A loop check L is *sound* if for every program P, goal G and SLD-tree T of $P \cup \{G\}$: if T contains a successful branch with computed answer substitution σ, then $f_{L(P)}(T)$ contains a successful branch with a computed answer substitution σ' such that $G\sigma' \leq G\sigma$.

iii) A loop check L is *shortening* if for every program P, goal G and SLD-tree T of $P \cup \{G\}$: if T contains a successful branch D with computed answer substitution σ, then either $f_{L(P)}$ (T) contains D or $f_{L(P)}$ (T) contains a successful branch D' with a computed answer substitution σ' such that $G\sigma' \leq G\sigma$ and $|D'| < |D|$. ❑

Definition 2.6. A loop check L is *complete* if every infinite SLD-derivation is pruned by L. ❑

Since logic programs are computationally complete, i.e., they have the same power as recursive functions, and since a loop check is computable, any weakly sound loop check cannot be complete for all programs. In [2] it was shown that any weakly sound simple loop check cannot be complete for all function-free programs.

In the following, two simple loop checks and some their variants are recalled from [2], with their respective soundness and completeness properties. In the SLD-derivations, goals can be regarded as lists, so that both the number and the order of occurrences of atoms is important. The equality (subsumption) relation between goals regarded as lists is denoted by the symbol $=_L$ ($\subseteq_L$).

Definition 2.7. (Equality Checks for Goals) *Equals Variant/Instance of Goal*$_L$ check is the set of SLD-derivations

$EVG/EIG_L = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_0,\theta_0} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_{k-1}\backslash\theta_{k-1}} G_k)$

such that for some i, $0 \le i < k$, there is a renaming/substitution τ such that $G_k =_L G_i\tau$ }). ❑

Definition 2.8. (Subsumption Checks for Goals) The *Subsumption Variant/Instance of Goal*$_L$ check is the set of SLD-derivations

$SVG/SIG_L = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_0,\theta_0} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_{k-1},\theta_{k-1}} G_k)$

such that for some i, $0 \le i < k$, there is a renaming/substitution τ such that $G_i\tau \subseteq_L G_k$}). ❑

Definition 2.9. (Equality Checks for Resultants) *Equals Variant/Instance of Resultant*$_L$ check is the set of SLD-derivations

$EVR/EIR_L = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_0,\theta_0} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_{k-1},\theta_{k-1}} G_k)$

such that for some i, $0 \le i < k$, there is a renaming/substitution τ such that

$G_k =_L G_i\tau$ and $G_0\theta_0\theta_1\ldots\theta_{k-1} = G_0\theta_0\theta_1\ldots\theta_{i-1}\tau$}). ❑

Definition 2.10. (Subsumption Checks for Resultants) The *Subsumption Variant/Instance of Resultant*$_L$ check is the set of SLD-derivations

$SVR/SIR_L = RemSub(\{D \mid D = (G_0 \Rightarrow_{C_0,\theta_0} G_1 \Rightarrow \ldots \Rightarrow G_{k-1} \Rightarrow_{C_{k-1},\theta_{k-1}} G_k)$

such that for some i, $0 \le i < k$, there is a renaming/substitution τ such that

$G_i\tau \subseteq_L G_k$ and $G_0\theta_0\theta_1\ldots\theta_{k-1} = G_0\theta_0\theta_1\ldots\theta_{i-1}\tau$}). ❑

Theorem 2.1. (Soundness) The equality and subsumption checks for resultants are shortening and therefore sound. ❑

In [2, 3] three subclasses of programs have been identified and some completeness properties of the above loop checks were proved.

Subsumption checks are based on the inclusion of goals. Consequently, they are stronger than equality checks. However, equality checks are computationally more efficient. Moreover, in [6] it has been shown that Equality checks, when it is combined with a sound and complete modification of the SLD resolution (based on elimination of redundant atoms), is complete for the same classes of programs for which the Subsumption checks are.

In the paper, an implementation of the equality checks is presented. Such implementation benefits of the facilities provided by the PROLOG interpreter that is taken into account. Indeed, it exploits an high level control mechanism (Sleepers) [8] that allows to handle in a simple and practical way the history of the computation and makes possible an efficient implementation of the loop checking technique.

It is worth noting that the loop checks, given in the above definitions, compare every goal with every ancestor of it. Thus, the number of comparisons performed is quadratic in the number of generated goals. As a consequence such interpreter may be not very useful in pratice, since it spends more time in loop checking than in generating new goals. A technique (triangular loop check), which is able to reduce the number of comparisons while preserving the soundness and the completeness of the loop checks, is provided in [3]. According such a strategy, two goals G_i and G_k are compared if and only if $i < k$ and i, k are *triangular* numbers. A number d is called triangular if $\sqrt{1+8d}$ is an integer. The number of comparisons performed by a triangular loop check is linear in the number of generated goals. In Section 7 the implementation of triangular loop check is considered.

3 The Sleepers

Sleepers technique is a general, software-oriented approach, which provides the possibility to have complete access to low level structures of the host language. This important benefit is offered without giving up a high level programming style. This feature is a concrete aid to design, implement, and maintain complex software systems. The generality of the mechanism has been proved by several programming languages (Pascal, C, Common Lisp, Prolog) which the sleeper mechanism has been successfully embedded. The functionality of the mechanism has been tested during the realization of different softwares, such as programming environment [9], fast compilers [10], multi-paradigms programming language frameworks [11], constraint resolution systems [4]. In this section we try to give a fresh look of this mechanism, reminding the reader to [10] for a more complete discussion. The code fragments illustrated in this paper, refer to a C-like implementation of the sleepers.

The concept at the basis of the sleeper can be so synthetized. During the execution of a program, there exist a number of suspended, but active, sub-programs (function, procedure, predicates, etc.) calls of a running program. This situation occurs very frequently (recursive calls, called-caller couple, etc.) and provokes the "freezing" of the contexts created for each

of the suspended calls. Sleepers are chunks of code associated with a suspended context. This code remains sleeping, until a specific stimulus occurs. In such situation, the sleeper is awakened, independently from the sub-program that remains inactive in the run-time stack. Once awakened, the sleeper has reading/writing faculties in the environment created for the corresponding sub-programs. We report in the next paragraph the basic constructs that will be used in our discussion.

Some Basic Constructs

```
scriptof sleeper-name is script-name(par-list){ <statement>* }
```

`scriptof` associates the script *`script-name`* with the sleeper *`sleeper-name`*. A script is skilled to accomplish a particular task in the related context. Several scripts may be introduced for each context by means of different labels. The bridge between the past and the future of a computation is given by the binding of the parameters provided by the process that awakes the instances of sleepers, and the arguments present in the context created at the execution of the scriptof declaration. When in the body of a script we invoke the awakening of the actual sleeper we activate the previous (considering the creation time) sleeper present in the computation history; otherwise the control returns to the statement which follows the awakening statement (the `alarm` construct introduced herein after).

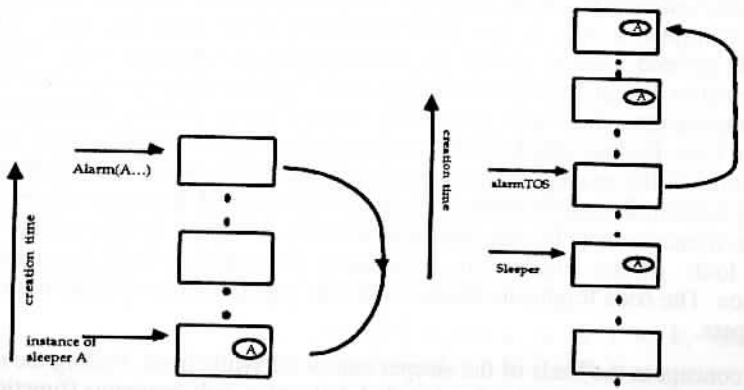
```
introduce script-name [,script-name]* during <statement>
```

`introduce` creates a new instance of the sleeper into the current context.

```
alarm(sleeper-name,arg*)
```

`alarm` performs the awakening of sleepers. The argument(s) arg* are passed to the selected instance.

The `alarm` executed in the current context switches the control in a previous block in which we inserted the sleeper. Several kinds of alarm-based construct are available. In the next figure we depict the difference of behaviour between the "standard" alarm and the `alarmTOS`, a variation of the above-defined construct.



When we execute an alarmTOS form, the search of the sleeper to awake starts from the last contexts, the youngest one present from the top of the run-time stack.

4 A Sleeper-Based Prolog System

In [10] it is shown how it is possible to implement a complete and efficient Prolog system (interpreter and compiler) thanks to the sleeper mechanism. The basic idea of this implementation, is to merge the main Prolog resource, the local stack, into the run-time stack of the host language. All the main activities of a Prolog evaluator are embodied in a unique function, **`evaluate`**, sketched in the next code.

```
evaluate(tobject *goal, tobject *env-goal)
   tobject *current-goal, *env-current-goal, …;
   tstatus  status;
   scriptof Backtrack is Down()       { … }
   scriptof Continuation is Up()      { … }
   scriptof Trail is Save(…)          { … }
{
   if null(goal) alarm(Continuation); /* a terminal point */
   current-goal = The-current-goal(goal, env-goal);
   switch (type_goal(current-goal)) {
      case built-in? : …;
      case prolog-r? :
         env-current-goal = The-env-current-goal(goal, env-goal);
               initialize(after-unif-env, trailed, … );
               available-rules = set-rules(current-goal);
            RESTART:
               introduce Save during
               status=unify(available-rules, after-unif-env
                        current-goal, env-current-goal, current-age);
               switch (type_unif(status)) {
                  case fail : alarm(Backtrack);  /* backtrack */
                  case choice-point :   /* a choice point CP*/
                           introduce Down, Save, Up during
                              evaluate(body(status), after-unif-env);
                  default :  introduce Up during /* not a CP */
                           evaluate(body(status),after-unif-env);  }
   }}
```

The sequence of contexts created to the call of **`evaluate`** corresponds to a Prolog computation history, which is managed in a transparent and simple way by the sleeper entities. The mentality to adopt in order to build a sleeper-oriented Prolog system is very simple. The context of evaluate provides all the resources important to handle the proof (some of these data are the list of the goals to prove, the trailed variables, the local environment, etc.) whereas the control activities are shared by specialized sleepers, such as the sleepers Continuation, Backtrack, Trail. For each created Prolog context, we allocate resources necessary for the demonstration. These ones are introduced and initialized by the function `initialize`. The unification of the current goal takes places by calling the function `unify`. Among its arguments we note: the current goal, its environment, the available rules, the age of the current agent and the environment of the unification process. The execution of this function modifies these resources and prepares the next of the demonstration.

Then we analyze the state of the unification. Three possible situations are examined:

•unification failed: we backtrack to the previous choice block. This task is performed by the specialized sleeper `backtrack`.

•unification succeeded (non deterministic state): the related parasites are introduced and the forward process goes on.

•unification succeeded (deterministic state): only an instance of the sleeper Continuation is introduced and the forward process goes on.

Here follows a brief description of some of the most significant sleepers.

Continuation. We insert this sleeper each time that there exist goals following the demonstration of the current atom.

Trail. Giving an intuitive idea, the existence in the evaluate contexts of such sleepers is equivalent to the existence of several *local* trail-based structures.

Backtrack. Each time that an evaluate context corresponds to a choice point, an instance of this sleeper is introduced in the corresponding context.

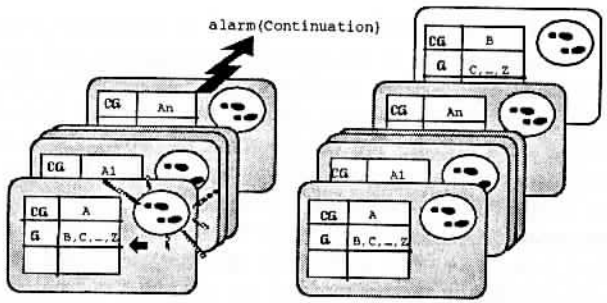
Now, let us focus our attention on the Forward mechanism, realized by the sleeper Continuation. Here follows the implementation of such a sleeper.

```
scriptof Continuation is Up()
   {
      if null(goal) alarm(Continuation);
      else evaluate(goal, env-goal);
   }
```

After any unification step, we introduce an instance of this sleeper Continuation in the current context (whether deterministic or not) to remember the existence of other goals not yet proved. After the unification, evaluate is recursively called with two new arguments: the body of the unified rule (as the new goals to be proved), and the environment created by the unification. If we have reached a terminal node, then the condition `null(goal)`, at the beginning of the body of evaluate, is satisfied. This state provokes the sending of an `alarm(Continuation)` through the demonstration history. This alarm will be propagated up to a context containing the goals which we have not yet proved. This situation is depicted in the next figure.

Let A, B, ..., Z be the goals with the rules:

(1) A:- A1,...,An.
(2) B:- B1,...,Bm.
A1.
A2.
....
An.



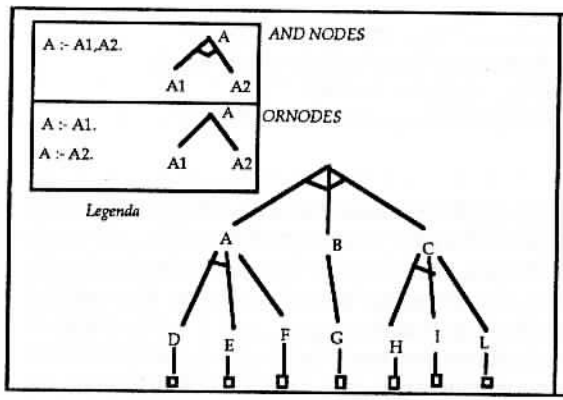
When we initially call evaluate with A, B, ..., Z as goals, an instance of the sleeper Continuation, depicted by ☹ is introduced in this block, with current goal CG A. After having completely demonstrated the body of the rule (1), we continue the proof, by revisiting the block with CG A in order to determine the next goals to be proved. This is done by awakening Continuation after the successful proof of the fact An. This action

leads to a call to evaluate with B, C, ..., Z as the new goals to be proved. The unification of B succeeds with the head of the rule (2). A new instance of the sleeper Continuation is introduced in the block with CG B. It will be alarmed after the successful proof of Bm.

Backtracking management is very simple. The sleeper is associated with choice context. When backtracking occurs, it is enough to alarm this sleeper. In such a way the control comes back in the last choice point. The awakening of this sleeper provokes the resetting of the environment and the elimination of the (usefulness) contexts, as the corresponding code shows.

```
scriptof Backtrack is Down()
{
   reset_vars(trailed); /* trailed variables are set to free */
   goto RESTART;     /*all the context from the top of the stack */
                     /* are disposed/*
}
```

More accurated details on the behavior of the above-mentioned sleepers can be found in [10]. To illustrate the policy of our interpreter in building the derivation tree, we show on the left side a synthetical representation of the tree and on the right side the entities responsible of the proof *activation*.



proof due to recursive call to evaluate	A-D-G-H
proof due to alarm continuation	E-B-C-I
proof due to alarm backtrack	L-F

Now let us face the problem of introducing a loop checking as embedded capability of the interpreter. To show how the sleepers enable to solve in an efficient and elegant way this problem, we prefere to adopt a progressive approach, by providing firstly the implementation of the Equality Check for Goals that are weakly sound loop checks.

5 Equality Loop Checks For Goals (Eig/Evg)

In this section we show the implementation choices undertaken to embed in the heart of our interpreter the management of EIG and EVG mechanisms. Let us start by providing the new version of the interpreter which supports such mechanism.

```
evaluate(tobject *goal, tobject *env-goal)
   tobject *current-goal, *env-current-goal, ...;
   tstatus  status;
   scriptof Backtrack is Down()
   scriptof Continuation is Up()
   scriptof Trail is Save(...)
   scriptof LoopCheck is EqInGo(…)
if null(goal) alarm(Continuation)
current-goal = The-current-goal(goal, env-goal);
switch (type_goal(current-goal)) {
   case built-in?: ...;
   case prolog-r? :
      enV-current-goal = The-env-current-goal(goal, env-goal);
      initialize(after-unif-env, trailed, ... );
      available-rules = set-rules(current-goal);
   RESTART:
      introduce Save during
      status=unify(available-rules, after-unif-env
          current-goal, env-current-goal, current-age);
     switch (type_unif(status)){
        case fail : alarm(Backtrack);
        case Loop : alarm(LoopCheck, goal,env-goal);
        case choice-point :
              introduce Down, Save, Up, EqInGo during
                    evaluate(body(status), after-unif-env);
        default :   introduce Up, EqInGo during
                    evaluate(body(status),after-unif-env);  }
   }
}
```

The implementation idea is very simple. The reader can note that only little modifications have been necessary to extend our interpreter to EIG treatment. Essentially, the main change consisted in inserting an additional sleeper, named `LoopCheck`, designed to handle the loop check calls. The awakening of this sleeper occurs after a derivation step without success (the value `Loop` returned from the boolean function `type_unif` on the status of the last unification). As arguments of such alarm, we have the goal list (the parameter `goal`) and the related environment (the parameter `env-goal`). This command has the effect to awake the last instance of such sleeper, suspended in the history of the calculus. As the reader can remark, we introduce an instance of such sleeper in each context of evaluate, independently form the fact of being a deterministic or not context. Before focusing on the behavior of the sleeper `LoopCheck` we provide the code implementation of such sleeper

```
/* Loop Check: Equality Instance */
scriptof LoopCheck is EqInGo (tobject *fgoal, *fenv)
   {
   if PG_unify(fgoal,fenv,goal,env-goal)
      alarmTOS(Backtrack) /* EIG_L found! */
   else alarm(LoopCheck,fgoal,fenv))
}
```

This code implements the loop check mechanism of Equality Instance of Goal. At the activation of the script `EqInGo` two objects are compared: the last goal to prove (the

parameter `fgoal`) and the previous one, `goal`, now accessible in the current, old context thanks to the sleeper `LoopCheck`. The analysis is performed in the corresponding environments and is carried out by the function `PG_unify`. The behavior of such function is partially based on that of a classical unifier, extended to additional features in such a way to provide the substitution/rename τ that makes G_i equal to G_k, e.g. $G_i\tau = G_k$. If this control succeeds, then we invoke the alarming of a backtracking, starting from the top of the run-time stack, and not from the current evaluate context (the special `alarmTOS`, used to alarming from the top of stack). In fact, the control returns to the first usable choice point, created before G_k. If the unification-like analysis of the two patterns of goal fails, then we continue to investigate in the past steps of the demonstration, by alarming still the sleeper `LoopCheck`. If all the evaluate contexts have been visited, then the activation of a last sleeper `LoopCheck`, associated to a default evaluate context, returns the control to the current context, which originated the search of a loop check.

If we want to handle loop check mechanism for Equality Variant of Goal, then it is necessary to associate to the sleeper LoopCheck a different script, as shown in the following code.

```
/* Loop Check: Equality Variant */
scriptof LoopCheck is EqVaGo(tobject *fgoal, *env-fgoal)
   {
   if not PG_unify(fgoal,fenv,goal env-goal)
       alarm(LoopCheck,fgoal,fenv))
   else if PG_unify+(fgoal,fenv,goal env-goal)
        alarmTOS(Backtrack) /* EVG_1 found! */
        else alarm(LoopCheck,fgoal,fenv))
}}
```

Firstly we verify if the test PG_unify fails. In fact, in this case we are sure that in this context the Equality Variant Status (EVS) does not occur and so, we can go down in other past proofs. If the test PG_unify ends with success, we analyze the two patterns of goals by means of the function `PG_unify+`, which is a finer version of the unification-based algorithm implemented by the `PG_unify`.

6 Equality Loop Checks For Resultants (Eir/Evr)

This section is dedicating in discussing the implementation for the management of EIR/EVR loop check mechanisms. As usual, we start by reporting the code of the `evaluate`:

```
evaluate(tobject *goal, tobject *env-goal)
   tobject *current-goal, *env-current-goal, ...;
   tstatus  status;
      tobject  *G0teta1_k,  *G0teta1_i,  *G0teta1_iTau,  ...;
   scriptof Backtrack is Down()
   scriptof Continuation is Up()
   scriptof Trail is Save(...)
   scriptof  LoopCheck  is  EqInRe(…)
if null(goal) alarm(Continuation);
current-goal = The-current-goal(goal, env-goal);
...;
G0teta1_i  =  alarm(PickG0);
switch (type_goal(current-goal)) {
   case built-in?: ...;
   case prolog-r? :
      env-current-goal = The-env-current-goal(goal, env-goal);
      initialize(after-unif-env, trailed, ... );
      available-rules = set-rules(current-goal);
   RESTART:
      introduce Save during
      status=unify(available-rules, after-unif-env
          current-goal, env-current-goal, current-age);
     switch (type_unif(status)){
        case fail : alarm(Backtrack);
        case Loop : alarm(LoopCheck, goal,env-goal);
        case choice-point? :
              introduce Down, Save, Up, EqInRe during
                    evaluate(body(status), after-unif-env);
        default :   introduce Up, EqInRe during
                    evaluate(body(status),after-unif-env);  }
   }}
```

The main difference between the goal-based and resultant-based equality checks implementation is the condition $G_0\theta_0\theta_1...\theta_k = G_0\theta_0\theta_1...\theta_i\tau$ to prove each time that a sleeper `LoopCheck` is awakened. Let us discuss what it means, from the standpoint of the implementation, the treatment of the previous condition. Essentially, as the reader can note, the alarming of the sleeper happens as for the EIG/EVG cases. Let us provide the script details:

```
/* Loop Check: Equality Instance */
scriptof LoopCheck is EqInRe (tobject *fgoal, *fenv)
   {
   tau = PG_unify(fgoal,fenv,goal,env-goal)
   if not tau alarm(LoopCheck ,fgoal,fenv)
   else {
       G0teta1_k= alarm(PickG0);G0teta1_iTau= AddTau(G0teta1_i,tau);
       if Eq(G0teta1_k,G0teta1_iTau) alarmTOS(Backtrack)
       else alarm(LoopCheck ,fgoal,fenv))}
}
```

The behaviour of the script is now more complex. The first action consists in calculating the renaming/substitution (the parameter `tau`). If such value exists, then we continue the exam of the current context of evaluate, otherwise we can re-awake the other sleepers `LoopCheck`.

If the analysis of the context goes on, then it is necessary to build two fundamental objects: $G_0\theta_0\theta_1...\theta_k$ and $G_0\theta_0\theta_1...\theta_i\tau$. The first value is allowable directly inside the context created for the proof of G_0. In fact, only in this context (the one created for the top-level goal), we have inserted a special-purpose sleeper, named `PickG0`, whose task is to return the value $G_0\theta_0\theta_1...\theta_k$. Here follows the description of this sleeper.

```
scriptof PickG0 is giveG0inEnv () {return(Instance(goal,env-goal))}
```

Now we must compute the value $G_0\theta_0\theta_1...\theta_i\tau$. This task is accomplished in two steps. Firstly we obtain the value $G_0\theta_0\theta_1...\theta_i$ (the parameter `G0teta1_i`). This value is computed at each demonstration step via the sleeper `PickG0` (see the code of the `evaluate`). The script `EqInRe` updates this local resource on the basis of the value `tau` (the renaming/substitution) just computed upon the unification-based analysis between the last goal G_n and the current one G_i. Finally we compare the two objects `G0teta1_k`, and `G0teta1_iTau`; if they are equal we establish a loop check status and the corresponding search of a choice point can start, otherwise we continue to explore the past derivation steps by alarming the sleeper `LoopCheck`.

7 Optimization

The proposed realizations are based on the naive principle to examine all the contexts in which there is the sleeper LoopCheck. Different important improvements must be adopted to enhance the implementation. One of the most important optimization regards the implementation of the triangular loop check, introduced in §2. The modifications to apply to the evaluate function are very simple, as shown in the next crunch of code:

```
            case Loop : alarm(LoopCheck, goal,env-goal);
            case choice-point? :
               if integer (sqrt(1+ 8*current-age)) then
                  introduce Down, Save, Up, EqInRe during
                     evaluate(body(status), after-unif-env)
               else
                  introduce Down, Save, Up, during
                        evaluate(body(status), after-unif-env);
               default :
                     if integer (sqrt(1+ 8*current-age)) then
                        introduce Up, EqInRe during
                           evaluate(body(status),after-unif-env)
                     else
                        introduce Up during
                           evaluate(body(status),after-unif-env);  }
...}
```

8 Conclusion And Future Work

The exigency to access to old contexts, suspended in a computation flow represent a good reason to use sleepers. The realization presented in this paper are characterized by a high-level description and by a simplicity of the code. No brutal intervention is noted in our realization. The most difficult tasks, which consist in the handling of data disseminated in

past contexts of a Prolog proof, are easily solved by a specialized sleeper, `LoopCheck`, designed to treat these particular situations. The implementation philosophy leads to a abstract strategy of incremental code production, which supports enhancements without considerable costs on the software. As future work, we are currently designing a Prolog compiler able to treat loop checks. Following an interesting trend for the Prolog compilers [12], we intend to compile into a high level language, in our case the C language enriched with the sleeper mechanism, rather than adopting the abstract machine technique.

References

[1] K.R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science* , volume B, pages 493-574. Elsevier, 1990.

[2] R.N. Bol, K.R. Apt and J.W Klop. An Analysis of Loop Checking Mechanisms for Logic Programs. In *Theoretical Computer Science* , volume 86, pages 35-79, 1991.

[3] R.N. Bol. Loop Checking in Logic Programming Ph.D. Thesis. Centrum voor Wiskunde en Informatica, Amsterdam, NL, 1991.

[4] C. Codognet, P. Codognet, V. Loia, and M. Quaggetto. Sleepers: a versatile high-level control mechanism. In *Proc. 1994 International Conference Programming Language Implementation and Logic Programming*, Madrid, Spain, September 1994, Spinger-Verlang LNCS Press.

[6] F. Ferrucci, G. Pacini and M.I. Sessa. Redundancy Elimination and Loop Checks for Logic Programs. In *Information and Computation* , to appear .

[7] J.W. Lloyd. Foundations of Logic Programming. Springer-Verlag, Berlin, second edition, 1987.

[8] V. Loia. Description opératoire d'interprètes Prolog complets en terme de manipulation de l'histoire de la démonstration. *PH.D Thesis,* Université Paris 6, France, 1989.

[9] V. Loia, M. Quaggetto, and F.-X Testard-Vaillant. A Prolog debugging system is an open system. *Workshop of the IV International Conference Logic Programming*, in Tech. Report ECRC IR-LP-31-25, Eilat, Israel, June, 1990.

[10] V. Loia, and M. Quaggetto. High level management of computation history for the design and implementation of a Prolog system. In *Journal of Software—Practice and Experience*, vol. 23(2), pp. 119-150, February 1993.

[11] V. Loia, and M. Quaggetto. Integrating Object-Oriented and Logic Programming: the OPLA Language. In *Proc. Ninth International Conference Knowledge Based Software Engineering*, Monterey, CA, USA, September 20-23, 1994, IEEE Press.

[12] J.L Weiner and S. Ramakrishnan. A Piggy-back Compiler for Prolog. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* Atlanta, Georgia, June, 1988, pp. 288-286.

Non Homomorphic Reductions of Data Structures

Luis Antonio Galán, Manuel Núñez, Cristóbal Pareja, Ricardo Peña

e-mail: {lagalan,manuelnu,cpareja,ricardo}@dia.ucm.es

Departamento de Informática y Automática

Universidad Complutense de Madrid

fax: (34-1) 394 4607 ph: (34-1) 394 4429

E-28040 Madrid. Spain

Abstract

In this paper we study different kinds of *reductions* of data types. By reduction we mean applying the higher order function *fold* to a data structure. An appropriate *fold* function can be defined for any recursive data type. These reductions have been presented as homomorphisms by several authors [2, 6]. Although many useful functions on data structures can be programmed as instances of *fold*, there are some that cannot. This is due to the fact that they are not mathematical homomorphisms. We show some examples of these functions. Then, we introduce two generalizations of *fold* (one for lists and the other for binary trees) in terms of which many non homomorphic mappings can be defined. Some examples are presented.

A second problem addressed in the paper is the relationship between the definitions of some particular reductions in different data types. We show that the definition of a particular reduction, e.g. to insert an element in a data structure, in terms of *fold* (either the generalized version or the usual one), looks the same for different data structures. In many cases, one can be obtained by transforming the other. We define a hierarchy of data types according to the amount of "structural" information they possess. It is shown how some reductions of a data structure A with more structural information than another one B can be obtained by composing the homomorphism from A to B that "forgets" structural information, with the homomorphism reducing B.

1 Introduction

There are two ideas that have been frequently stressed in transformational functional programming: the use of object-free functions [1] and the definition of a small set of higher order functions (*map, map2, foldr, filter*, etc.) that can cope with a large number of situations (e.g. the Bird-Merteens formalism [2]). Modern functional languages such as SML, Miranda, Haskell and Gofer [8, 7, 4, 5] are examples of this. The advantage of using these functions is twofold: in one hand, they encapsulate well