Integrative Functional Statistics in Logic Programming

Nicos Angelopoulos[1,2], Vítor Santos Costa[3,4], João Azevedo[5], Jan Wielemaker[6], Rui Camacho[5], and Lodewyk Wessels[1,2]

[1] Bioinformatics and Statistics, Netherlands Cancer Institute, Amsterdam, Netherlands
n.angelopoulos@nki.nl
[2] The Netherlands Consortium for Systems Biology (NCSB)
[3] CRACS-INESC Porto LA, Universidade do Porto
[4] DCC-FCUP, Universidade do Porto
Rua do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
vsc@dcc.fe.up.pt
[5] LIAAD & DEI & Faculdade de Engenharia, Universidade do Porto, Portugal
[6] Vrije Universiteit Amsterdam, Netherlands

Abstract. We present *r..eal*, a library that integrates the R statistical environment with Prolog. Due to *R*'s functional programming affinity the interface introduced has a minimalistic feel. Programs utilising the library syntax are elegant and succinct with intuitive semantics and clear integration. In effect, the library enhances logic programming with the ability to tap into the vast wealth of statistical and probabilistic reasoning available in *R*. The software is a useful addition to the efforts towards the integration of statistical reasoning and knowledge representation within an AI context. Furthermore it can be used to open up new application areas for logic programming and AI techniques such as bioinformatics, computational biology, text mining, psychology and neuro sciences, where *R* has particularly strong presence.

1 Introduction

Logic programming provides a powerful framework for reasoning with complex, structured data and is an important vehicle for AI research. The Prolog language is a popular example of logic programming that provides a query driven inference mechanism. Prolog has been shown to be useful in diverse AI application domains, including machine learning, natural language, data-base interfacing, web services, and program analysis. Often, Prolog applications require computing aggregate properties of data. Common operations, such as computing the mean or standard deviation, can be easily programmed in Prolog. More complex operations, such as clustering, pattern extraction or likelihood computations can be hard to implement efficiently.

The *R* environment [15] is an open source software package for statistical data analysis. *R* is widely used by the statistical and data mining communities, with major applications in areas such as bioinformatics. The *R* environment provides a set of effective tools for data storage and manipulation, namely of arrays, and it implements a well developed programming language, *S* [5]. Although *S* is not a pure declarative language, it contains a strong functional programming component. *R* also has an excellent packaging and distribution system through which a multitude of researchers and programmers make

K. Sagonas (Ed.): PADL 2013, LNCS 7752, pp. 190–205, 2013.

their code available to the community. The comprehensive *R* archive network (CRAN, `http://cran.r-project.org/`) contains a vast selection of contributed code that deals with the full gamut of statistical inference and data analysis. Examples include several implementations of the pagerank algorithm [8], machine learning tools such as support vector machines [10] and several clustering tools [13].

Prolog has been the main practical vehicle of logic programming with a number of open source implementations available to the community. *YAP* [7] and *SWI* [21] are two such systems. The first is widely accepted as one of the fastest open source Prolog implementation in, among other areas, machine learning and particularly in inductive logic programming (ILP) [18]. Machine learning is one application where it is natural to interface a Prolog system to *R*, namely to extend the ILP paradigm with regression capabilities [1]. The *SWI* Prolog system is the most widely used open source Prolog with many educational, research and industrial installations. It is well regarded for its stability and extensive palette of libraries. The software presented here has been developed to run on both of these Prolog systems, and is made available as open source software in the hope that it can be widely adopted and become a standard that enhances Prolog's capabilities.

A further motivation for integrating *R* with logic programming stems from the observation that traditionally, work on logic programming has focused in representing crisp knowledge. More recent work on the interplay between knowledge representation and statistical inference has attracted substantial interest in recent years. Work in this area includes the *PRISM* system and its EM-based parameter algorithm [16], Stochastic logic programs with an MCMC structure learning system [2] and the FAM algorithm [9], the ProbLog language and system [12] with a variety of learning algorithms and CLP($\mathcal{BN}$) [6], with EM learning and an interface to *Aleph* [18]. The interface to *R* allows the integrated statistical-logic inference systems access to a wide range of tools from random number generators to sophisticated algorithms for probabilistic inference.

Our work, the *r..eal* library, overcomes significant shortcomings present in earlier attempts to interface Prolog to *R* [1,4]. The first approach, *YapR*, used the *C* interface to pass *R* commands as sequences of characters with little conversion. In the second approach, *r_session*, the interface provided an expression based communication via an independent *R* process to which the operating system channelled I/O from Prolog. This approach was more flexible, but inefficient and hard to maintain across operating systems. The new approach, *r..eal*, introduces a completely new design that provides a tightly integrated interface to *R* for the Prolog programmer. In our approach *R* is invoked as a shared operating system library while the communication of large data between Prolog and *R* is facilitated by *C* code utilising the *C*-interfaces for the two systems.

We argue that critical to Prolog's success as a vehicle for AI research is its ability to address statistical aspects of knowledge representation and reasoning. Consider one domain which is currently experiencing a rapid expansion: computational biology. In this domain, vast volumes of data need to be interpreted and resulting knowledge to be represented. *R* packages such as *Bioconductor* [11] are among the most successful tools in this area, but they lack the knowledge representation strengths of Prolog. Thus, by combining logic programming with the extensive statistical functionality of *R*, we hope

to contribute to progress in this field while engaging more of the logic programming community in this area of research.

The rest of the paper is organised as follows. Section 2 presents the developed interface. Data representation in *R* and Prolog and the translation process is described in Section 3. Section 4 shows some illustrative examples and the conclusions are summarised in Section 5.

2 Interface

R..eal enables the communication between the Prolog system and *R*. The *R* environment executes as an operating system library: from the Prolog point of view, *R* is just another set of functions; from the *R* point of view, Prolog is the top-level. The user interface is designed to satisfy the following requirements:

- *Minimality:* ideally, most interactions should be performed through a small number of predicates.
- *R Flavour:* using the interface should be as close as possible to the standard usage of *R*. It should feel as if we are writing *R* code. To do so, most common *R* constructs should just work.
- *Prolog Flavour:* the interface should not require the user program to construct a sequence of characters to be interpreted by *R*. Instead, it should be about Prolog terms that are constructed and manipulated by Prolog code.

Arguably, the two last goals are incompatible, given the conceptual and syntactic differences between Prolog and *R*. *R..eal* tries to be as close to *R* as possible, but respecting the observation that ultimately one has to construct a valid Prolog program.

The library leaves the management of *R* variables to the programmer. On backtracking there is no removal of variables from the *R* environment. In practice, this is rarely a limitation, particularly since *R* variables can be destructively assigned new values. In our experience, the strengths of Prolog search through solutions spaces, merge well with a sequential application of R functions that can provide statistical computations.

2.1 Access Predicates

The *R* language uses `<-` as the assignment operator. In order to be as close to possible to *R* syntax, *r..eal* uses `<-/1` and `<-/2` to channel the bulk of the interactions between Prolog and *R*. The predicate names are defined as prefix and infix operators, respectively. The `<-/1` predicate sends an *R* expression, represented as a ground Prolog term, to *R*. The `<-/2` operator facilitates bi-directional communication. If the left-hand side is a free variable, the library assumes that we are passing data from *R* to Prolog. If the left-hand side is bound, *r..eal* assumes that we are passing data or function calls to *R*. The library implements two communication mechanisms:

- arbitrary *R* expressions of function calls which possibly embed data items within their arguments, are transformed from Prolog terms to strings and passed to *R* for native parsing

– Prolog lists and *R* vectors are passed by *r..eal* through *C* code that understands how Prolog and *R* represent data;

More concretely, the calling modes for `<-/2` are:

```
+Rvar   <- +PLvalue
-PLvar  <- +Rvar
-PLvar  <- +Rexpr
+Rexpr1 <- +Rexpr2
```

In the first, top-most mode, the *C* interface is employed to transfer Prolog data value(s), `PLvalue`, to an *R* variable identified by `Rvar`. In the second mode, *r..eal* instantiates the Prolog variable `PLvar` to the contents of the *R* variable `Rvar`. In the second mode `Rexpr` is evaluated in *R* and its result is unified to Prolog variable `PLvar`. In the current implementation this is done by first assigning the result to a hidden *R* variable and then using the second mode to copy this onto `PLvar`. In the last mode, *r..eal* will pass `Rexpr1 <- Rexpr2` to *R* subject to the syntactic conversions described in the next section. *R..eal* will automatically distinguish between the four modes. A variable in the left side of the operator is taken to be a Prolog variable, an atom is recognised as an *R* variable (`Rvar` above) and a ground term is considered to be an *R* expression. On the right side, a list or a number are taken as Prolog data, an atom corresponding to a known *R* variable is recognised as such and all other terms are *R* expressions.

In the following example a list of 6 Prolog integers is passed to the *R* variable `v` and then passed to Prolog variable `V`.

```
?-  v <- [0,1,1,2,3,5],
    V <- v.

V = [0, 1, 1, 2, 3, 5].
```

In the arity 1 version of the assignment predicate, if the argument can be interpreted as a known *R* variable then it is printed using the *R* function call `print()`. The following example prints the contents of an *R* variable (`v`) that has been passed a list of Prolog integers.

```
?-  v <- [0,1,1,2,3,5],
    <- v.

[1] 0 1 1 2 3 5
```

When *r..eal* cannot establish that the argument of `<-/1` is an *R* variable, it passes the argument to *R* as an expression right after all syntactic transformations have been completed. This allows for calling of functions to which the return value is of no interest to the user. For instance the value of the plotting function is often ignored. The following example uses *R*'s `plot()` function to plot 3 points with x-coordinates $[1, 2, 3]$ and y-coordinates $[4, 5, 6]$. The plot appears on *R*'s default plot display.

```
<-  plot( [1,2,3], [4,5,6] ).
```

3 Data Representation in R

R recognises several types of objects:

- Floating point numbers, integers, Boolean and ascii values (character strings) provide the base types.
- Lists or vectors are the main forms of serialised compound objects.
- Arrays are multi-dimensional compound objects with two dimensional arrays treated as special arrays called matrices.
- *R* supports several useful data-types: dotted-pairs are used to represent lists; the `:` operator is supported for ranges, and `NULL` objects represent uninitialised *R* objects.
- Programs can be constructed by using *symbols*, functions or closures, and environments.

Regarding base types, there are matches between floating point and integers in *R* and Prolog. Boolean values can be matched to `true` and `false` atoms. Character strings are traditionally represented by Prolog as lists of character codes. These principles correspond to the following rules:

```
Prolog      ---  R
integer     <->  integer
float       <->  double
atom        <->  char
char         ->  char
true/false  <->  logical
```

The three other major types supported by the interface are symbols, vectors and matrices. Symbols are *R* identifiers used for variable and function names. They naturally map to Prolog atoms and they are contextually distinguished from chars. Compound objects are described in detail next.

3.1 Vectors and Matrices

Vectors are a key generic data type in *R*. It is important to make two observations on the nature of vectors in *R*. First, that *R* vectors are typed and second that they have attributes. *R* has six basic vector types: logical, integer, real, complex, string (or character) and raw. The other major data types in *R* include lists, expressions and functions. As an example, the *R* variable `v`, defined by

```
?-  v <- as..integer(c(1,2,3)).
```

is of type integer vector and its contents are the values $1, 2$ and 3. Note that `c()` is a generic method in *R*. The default function of this method is to combine its arguments into a vector. A vector naturally translates to a list in Prolog. Multi-dimensional arrays are mapped to lists of lists. This principle works both ways: Prolog lists are mapped to vectors, and lists of lists to matrices (which are 2 dimensional arrays in *R* parlance). *R..eal* provides two main ways to pass Prolog data to *R*. The more efficient method is

by using the *C* interface while the alternative method constructs a string representation of an *R* command. The former method is accessible to the user via the mode of `<-/2` in which Prolog data is passed to an *R* variable. Goals have the following form, where `PLvalue` is the Prolog data and `Rvar` is the *R* variable.

```
+Rvar <- +PLvalue
```

This mode is implemented in *C* and transfers via *C* data from Prolog to *R*. The type of values of the vector or matrix is taken to be the type of the first data element of the `PLvalue`. An example of passing a list of the integers between 1 and 100 to an *R* variable (`i`), printing the first ten elements through *R* and then passing the vector back to Prolog after adding 1 to each number follows:

```
?-  findall( I, between(1,100,I), Is ),
     i <- Is,
     <- i^[1:10],                % prints via R
     Js <- as..integer(i+1).

[1]  1  2  3  4  5  6  7  8  9 10
Is = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
Js = [2, 3, 4, 5, 6, 7, 8, 9, 10|...].
```

3.2 Object Types

When passing Prolog objects to *R*, *r..eal* attempts to build the *R* structure by extrapolating the object type from the type of the first element in the Prolog structure. If later on this breaks down, the structure is rebuilt if the type that introduced the failure can be used for the overall data structure.

For example, the Prolog list in the following query contains a float value in its third position. As Prolog is untyped, we do not have this information when the list starts being transferred across. Instead, at the start of passing the list through, the first element is inspected and the list is assumed to contain integers. At the third element, upon encountering a float value, the work done so far is scrapped and the more general type is used to translate the list to a vector of float values.

```
?-  r <- [1,2,3.2,4],
    <- r,
    R <- r.

[1] 1.0 2.0 3.2 4.0
R = [1.0, 2.0, 3.2, 4.0].
```

3.3 The Expression Mechanism

Data appearing in an arbitrary *R* expression is parsed and placed into a string that will then be passed from Prolog to *R* for evaluation. For instance, in the following example

the `c()` combinator function is used to combine 5 values into an *R* vector before printing it and then pasting all vector elements to a single value vector (`s`). For illustration purposes we also include a goal that combines the two function calls (assignment to *R* variable `t`).

```
?-  state <- c("tas","sa","qld","nsw"),
    <- state,
    s <-paste(state,collapse="+"),
    t <-paste(c("tas","sa","qld","nsw"),collapse="+"),
    <- s,
    <- t.

[1] "tas" "sa"  "qld" "nsw"
[1] "tas+sa+qld+nsw"
[1] "tas+sa+qld+nsw"
```

The implementation of *r..eal* recognises that the expression to be assigned to *R* variable t is not a single Prolog data term but a number of *R* function calls, so it transforms this expression into a string containing an *R* expression. Note that when using this interface it is convenient to represent *R* chars by Prolog list of codes, as in the above example.

Passing long objects through the expression mechanism is both inefficient and can easily lead to buffer limitations as it is only intended as a mechanism for passing function calls on existing *R* objects. *R..eal* circumvents both these limitations by automatically detecting Prolog lists and `c()` terms and passing them via a *hidden R* variable which is then substituted in the call passed for evaluation to *R*. The temporary name of the hidden variable is selected so as not to clash with the current *R* name-space.

For instance, the following code generates a list of $50,000$ elements and computes the mean of its elements via a call to *R* through the expression mechanism. Without the use of hidden variables this call would generate a resource error and even shorter lists would take much longer to transfer. The example code that follows was executed on SWI-Prolog 6.3.0 on a Linux 11.10 desktop having a dual core 3.16 GHz processor.

```
?-  findall(I,  between(1,50000,I), Is),
    time( A <- mean(Is) ).

% 181 inferences,0.002 CPU in 0.002 seconds
                                    (100% CPU,75597 Lips)
Is = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
A = 25000.5.
```

In the above calls, `A <- mean(Is)` becomes `t <- Is, A <- mean(t).`

3.4 Syntactic Issues

There are syntactic conventions in *R* which result in non-parsable Prolog code. Notably function and variable names are allowed to contain dots, square brackets are used to access parts of vectors and arrays, and functions are allowed empty argument tuples.

We have introduced syntax which allows for easy translation between Prolog and *R*. Prolog constructs are converted by the library as follows:

- *R* code often uses the '`.`' symbol with function and variable names. As this syntax conflicts with standard Prolog usage, *r..eal* allows the use of the operator '`..`', e.g.:

```
as..integer(c(1,2)) => as.integer(c(1,2))
```

  The library's name is a word play on the '`..`' operator.
- *R* allows matrix subscripts. In the style of BProlog [22], *r..eal* uses the '`^`' operator:

```
a^[2] => a[2]
```

- *R* allows ranges over subscripts, say `a[,,2]` which in *R* is a way of to refer to all the values of the first and second dimension of `a`. *R..eal* uses `*` for this purpose:

```
a^[*,*,2] => a[,,2]
```

  Note that *r..eal* follows *R* conventions to access arrays.
- We map the '`$`' *R* operator to a Prolog library operator (`op(400,yfx,$)`). In *R*, `$` is one of the possible ways in which parts of vectors, matrices, arrays and lists can be extracted or replaced. In most contexts there is no ambiguity so the operator can be used freely, however in some situations it might be necessary to quote.

```
a$val => a$val   or   'a$val' => a$val
```

- *R..eal* uses `(.)` to denote *R* functions with zero arity:

```
dev..off(.) => dev.off()
```

- The *R* `NULL` value is coded as the empty list.
- Simple *R* functions can be coded by using the Prolog implication operator '`:-`':

```
(f(x) :-  (...)) => f(x) (...)
```

  This is only advised for very small functions, and does not support conditionals yet.
- As mentioned previously, lists of lists are converted to matrices. In contrast to the flexibility of *R*, all levels of the lists must have the same length.
- Prolog represents character strings as lists of integers. It is thus impossible to distinguish strings from genuine lists of integers appearing in arbitrary *R* expressions. We define '+' as a prefix operator to identify strings.

```
source(+"String")  =>  source("String")
```

- Some *R* operators should be quoted in Prolog:

```
a '%*%' b  => a %*% b
```

The majority of *R* operators can be used unquoted as they are defined as infix operators and present no issues. Finally, expressions that *r..eal* cannot translate can always be passed as Prolog atoms or strings.

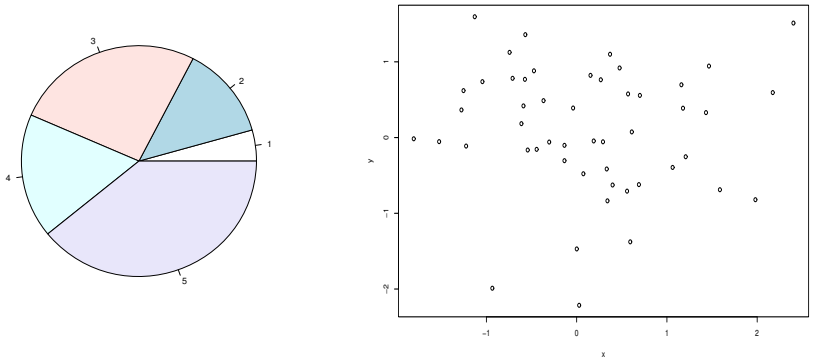

Fig. 1. Left: pie-chart example of a Prolog list of integers. Right: plotting two sequences of 50 random numbers as X and Y coordinates.

4 Applications

4.1 Plot Drawing

Visualising data is a particular strength of *R*, whereas Prolog systems traditionally have only limited access to graphics. The *r.eal* interface enables access to the extensive facilities of *R*. Simple plots such as scatter plots, histograms, box plots and pie charts can be easily drawn for Prolog data objects. In Figure 1 a pie chart and a scatter plot are shown. In the first example a list of integers is passed and plotted as a pie-chart, where each integer indicates the relative area of each slice. The following is the code for drawing the pie-chart shown in the LHS of Fig. 1.

```
?-  cars <- [1, 3, 6, 4, 9],
    <- pie(cars).
```

The next example, also from Fig. 1, shows how to create a plot of 50 random samples whose coordinates have been drawn from a normal distribution (`rnorm()`). The coordinates are stored in *R* variables `x` and `y` before being plotted on a new plotting window created with the `x11()` function. Over twenty different probability distributions are present in *R*, with more available in add-on packages.

```
?-  <- set..seed(1),
    y <- rnorm(50),
    x <- rnorm(y),
    <- x11(width=5,height=3.5),
    <- plot(x,y).
```

A third plotting example is shown in Figure 2. In this case, the nested outer product of two vectors defined implicitly using the column notation (`0:9`) is computed. The results are then tabled and plotted. Labels are passed to the plot via variables instantiated to character strings. The generating code is as follows:

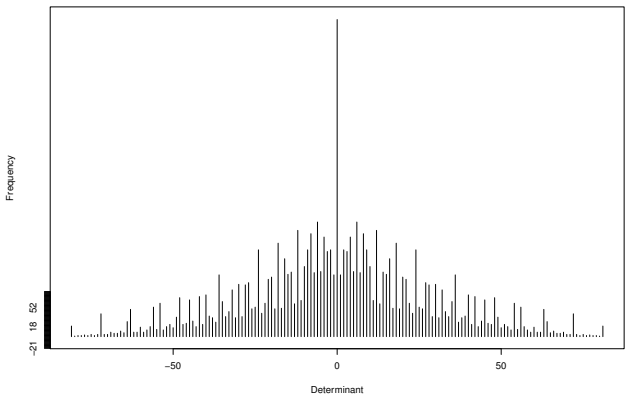

Fig. 2. Tabled nested outer product of $0:9$ to itself

```
?-  d <- outer( 0:9, 0:9 ),
    fr <- table(outer(d, d, +"-")),
    Xl = "Determinant", Yl = "Frequency",
    fr..names <- as..numeric(names(fr)),
    <- plot(fr..names,fr,type=+"h",xlab=+Xl,ylab=+Yl).
```

4.2 Interacting with Datasets

Interfacing to *R* allows access to a large variety of data formats. As an example, consider the comma-separated values (`csv`) format file `trees91.csv`:

```
C,N,CHBR,REP,LFBM,STBM,RTBM,LFNCC,STNCC,RTCACC,LFKCC,STKCC,...
1,1,CL6,1,0.43,0.13,0.29,1.84,0.4,0.96,0.13,0.06,0.23,0.3,...
1,1,CL7,1,0.4,0.15,0.25,1.82,0.37,0.95,0.18,0.06,0.22,0.22,...
1,2,A1,9,0.45,0.2,0.21,1.54,0.96,0.69,0.16,0.08,0.3,0.35,...
...
```

The first line contains headers, and the remaining lines contain data in a tabular format, separated by commas. Reading the file into an *R* variable (`tree`) is done by simply calling:

```
?-  tree <- read..csv(file="trees91.csv",
                                  sep=",",head='TRUE').
```

For instance, to get the column names in a Prolog list we can do:

```
?-  X <- names(tree).

X = ['C','N,'CHBR','REP'|...].
```

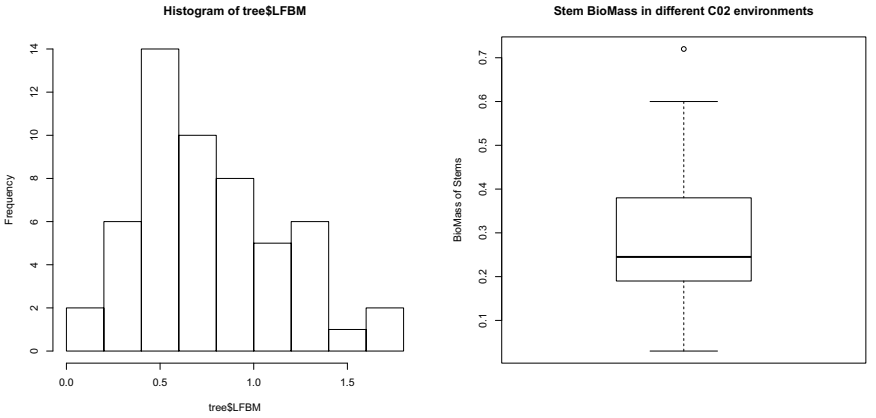

Fig. 3. Reading csv data. Left: histogram of `LFBM` column. Right: box plot of `LFBM` column.

The `$` operator is used to access columns of the read table:

```
?-  X <- tree$'LFBM'.

X = [0.43,0.4,0.45,0.82,0.52,1.32,0.9,1.18,0.48|...].
```

It is straightforward to obtain and plot a histogram of a specific column (LHS, Fig. 3):

```
?-  X <- hist(tree$'LFBM').

X = [breaks=[0.0, 0.2, 0.4, ...], ... ].
```

This will both fill X with the histogram and plot it in the graphical interface (RHS, Fig. 3). Plotting can be avoided when passing the FALSE value to the argument `plot`, while the histogram can also be plotted without any value being explicitly returned by using `<-/1`:

```
?-  X <- hist(tree$'LFBM', plot = 'FALSE').

X = [breaks=[0.0, 0.2, 0.4, ...], ... ].

?-  <- hist(tree$'LFBM').
```

The same plot can be saved in a PDF file. The function `pdf()` opens a new graphics device with output to the named PDF file, while the *R* function `dev.off()` closes the graphics device that was opened last.

```
?-  <- pdf(+"plot.pdf"),
    X <- hist(tree$'LFBM'),
    <- dev..off(.).

X = [breaks=[0.0, 0.2, 0.4, ...], ... ].
```

The data can be displayed in a variety of different formats, for example as box plots (RHS, Fig.3).

```
?- Main="Stem BioMass in different C02 environments",
   Y = "BioMass of Stems",
   <- boxplot(tree$'STBM', main=+Main, ylab=+Y).
```

4.3 Pagerank on Prolog Programs

We have so far seen how Prolog can command *R* and pass data to it. We can further observe that Prolog programs are term structures themselves, suggesting that we might want to apply a variety of well known statistical algorithms to the analysis of Prolog programs. The next example shows an application where we take advantage of the respective strengths of Prolog and *R*. The goal is to find the most important, or cross-referenced, procedure in a Prolog program by using a graph algorithm on a network representing the call dependencies of the program under investigation. We use the popular pagerank algorithm [14] and its implementation in *R*'s *igraph* package [8]. We apply our analysis on the source code of the ILP program *Aleph* [18].

The first building block of our program visits the Prolog source and constructs a graph where the nodes are the predicates used by *Aleph*. We define procedure `parse/2` to collect all edges in a source file:

```
parse(File, Nodes) :-
   open(File, read, S),
   findall(Node, clause_to_nodes(S, Node), Nodes),
   close(S).
```

The program scans every clause in the file. For each clause, it first maps the head and every sub-goal in the body to an integer corresponding to its defining predicate. Then, it creates an edge between every sub-goal and the head. As an example, in the following clause

```
subtract([E|T], D, R) :-
    memberchk(E, D), !,
    subtract(T, D, R).
```

the program will first map `subtract/3` to 0 and `memberhck/2` to 1 and then generate the graph $\{1 \mapsto 0, 0 \mapsto 0\}$. As *R*'s `graph()` function prefers to receive the graph as a list of nodes we make `clause_to_nodes/2` succeed four times with answers $Node = 1$, $Node = 0$, $Node = 0$, and $Node = 0$, so that two consecutive solutions represent an edge. Solutions are then captured by `findall/3` as a list that is passed on to the *R* environment and the `graph` function.

We define the `pagerank/1` procedure to find the maximum element of the graph nodes in a file as computed by `page.rank()`.

```
pagerank(File,nav(Name,Arity,Value)) :-
   parse(File,Graph),
   g <- graph(Graph),
```

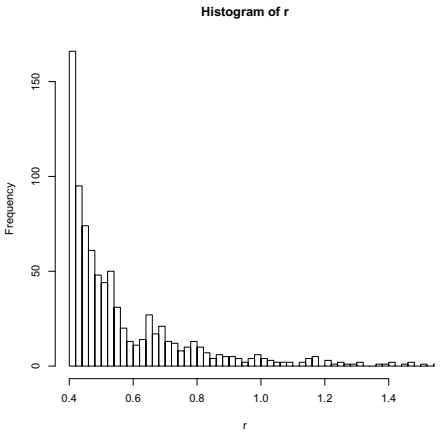

Fig. 4. Histogram of pagerank scores for *Aleph* predicates

```
r <- page..rank(g),
Scores <- r$vector,
max_element(Scores, Name, Arity, Value).
```

This predicate obtains the list of edges, and calls `graph()` to create a weighted graph `g`, where the weight is the number of repeated occurrences. It then computes the page rank scores into the *R* object `r` and reads the vector component of the object to list $Scores$. The `max_element/3` procedure simply extracts the predicate with highest pagerank.

Applying the program to ILP system *Aleph* generates a graph with 968 nodes and 7296 edges. The highest score in the graph is for `!`, which is unsurprising. If we remove all built-in predicates the highest score is for `$aleph_global/2`. We can also reverse the graph. In this case the highest score is for `reduce/0`. Figure 4 shows a histogram of pagerank scores for the predicates in *Aleph*.

4.4 Search and Visualisation

R and Prolog are complementary in that the former has strong presence in data analysis and visualisation while the latter has strengths in knowledge representation and search based reasoning. In order to underline this and point at computational biology and bioinformatics as important areas of applications, we employ *r..eal* as a bridge between a search algorithm implemented in Prolog and visualisation component via the *RCytoscape* [17] *Bioconductor* package.

The objective is to build a network of interactions between genes that encode proteins that are involved in cell motility. Direct edges, representing interactions within this set of genes (the *adhesome library*), are extracted from the Kyoto Encyclopedia of Genes and Genomes (KEGG). We then employ a breadth first algorithm to join disconnected adhesome genes to the main network by finding one of the shortest paths involving the

Fig. 5. KEGG interactions for a subset of the members of an adhesome library. Nodes are proteins and edges denote interactions. Blue nodes are connecting proteins that do not appear in the library. Edges are coloured as per type of interaction.

minimum number of intermediary genes that are not in the library. The results can then be displayed via the *RCytoscape* package which interfaces *R* to the *Cytoscape* biological network visualiser.

Figure 5 shows the graph of the adhesome library as searched by Prolog and displayed by *RCytoscape* via calls to *r..eal*. This example utilises an in-house library that uses *R* and its bridge to *Cytoscape* (*RCytoscapse*) to display arbitrary Prolog graphs (as those managed with the *u*graph library). We hope to publish this soon, inclusive of the code for this example. The connection established with *RCytoscape* is bi-directional. The user can use all facilities in *Cytoscape*, such as selecting nodes or edges. Lists of such selections can be queried via *r..eal* which can be a starting point for further analysis and search within Prolog. A more general discussion on Prolog and *r..eal* for bioinformatics can be found in [3].

5 Conclusions

The library presented here achieves a tight integration of the *R* statistical software system with two open source Prolog implementations. Our designing principles have been those of simplicity and transparency across the systems. This has been accomplished by (a) keeping to a minimum the transformations the user needs to be aware of, and by (b) providing intuitive, mnemonic syntax to the inconsistencies between the two languages. As a result, *r..eal* programs are clear and easy to follow. The functional inheritance of *R* corresponds well with the logical underpinning of Prolog. *R..eal* provides a productive environment for building highly effective pipelines and interactive, query-based data exploration.

Interfacing the *R* environment with Prolog widens the range of applications for logic programming and inductive logic programming. It has the potential to facilitate the development of systems combining logic and probabilistic reasoning and will significantly improve the development of ILP applications requiring statistical and numerical computations. We also hope that this interface will encourage logic programming researchers to engage in areas of research where a synergy of knowledge representation and statistical prowess is needed such as in bioinformatics and computational biology. Symmetrically, our library increases the tools available to *R* researchers and programmers who wish to exploit Prolog's advanced AI capabilities.

Possible extensions to the library include tighter integration with backtracking, although this has not been a limitation to the current applications. One specific aspect of such closer integration that might be of immediate value is the re-use of *hidden* variables (Section 3.3). An even tighter integration might be possible by allowing hidden and other *R* variables to be available for garbage collection. Finally, it would be interesting to investigate an even tighter syntactic integration by means of extensions to the syntax admitted by Prolog.

R..eal was originally designed, developed and tested on *YAP 6.3.1* under the Linux operating system. It has also been compiled for, and known to be working on MS operating systems and Mac OS. It was later ported [19] to the *SWI* [21] engine via a complete re-write of the *C* code. This has become the main development code as *YAP* provides a comprehensive compatibility layer to *SWI*'s *C* interface [20]. The library and examples presented here can be downloaded from our website (`http://bioinformatics.nki.nl/~nicos/sware/r..eal/`).

Acknowledgments. This work was co-financed by the Netherlands Consortium for Systems Biology (NCSB) which is part of the Netherlands Genomics Initiative / Netherlands Organisation for Scientific Research. VSC is funded by the ERDF through the Progr. COMPETE and by the Portuguese Gov. through FCT-Found. for Science and Tech., proj. LEAP ref. PTDC/EIA-CCO/112158/2009 and FCOMP-01-0124-FEDER-015008, and proj. ADE ref. PTDC/EIA-EIA/121686/2010 and FCOMP-01-0124-FEDER-020575.

References

1. Alves, A., Camacho, R., Oliveira, E.: Discovery of functional relationships in multi-relational data using inductive logic programming. In: IEEE Int. Conf. on Data Mining, pp. 319–322. IEEE Comp. Society, CA (2004)
2. Angelopoulos, N., Cussens, J.: Bayesian learning of Bayesian networks with informative priors. Journal of Annals of Mathematics and Artificial Intelligence 54(1-3), 53–98 (2008)
3. Angelopoulos, N., Shannon, P., Wessels, L.: Search and rescue: logic and visualisation of biochemical networks. In: Proceedings of the ICLP 2012 Workshop on Constraints in Bioinformatics (WCB 2012), Budapest, Hungary, pp. 1–6 (September 2012)
4. Angelopoulos, N., Taylor, P.: An extensible web interface for databases and its application to storing biochemical data. In: WLPE 2010, Scotland (July 2010)
5. Becker, R.A., Chambers, J.M., Wilks, A.R.: The New S Language: A Programming Environment for Data Analysis and Graphics. Wadsworth & Brooks/Cole, USA (1988)

6. Costa, V.S., Page, D., Qazi, M., Cussens, J.: CLP(BN): Constraint logic programming for probabilistic knowledge. In: Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI 2003), pp. 517–524 (2003)
7. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Journal of Theory and Practice of Logic Programming 12, 5–34 (2012)
8. Csardi, G., Nepusz, T.: The igraph software package for complex network research. InterJournal, Complex Systems 1695 (2006)
9. Cussens, J.: Stochastic logic programs: Sampling, inference and applications. In: Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI 2000), San Francisco, CA, pp. 115–122 (2000)
10. Dimitriadou, E., Hornik, K., Leisch, F., Meyer, D., Weingessel, A.: e1071: Misc Functions of the Department of Statistics (e1071), TU Wien (2011)
11. Gentleman, R.C., Carey, V.J., Bates, D.M., et al.: Bioconductor: Open software development for computational biology and bioinformatics. Genome Biology 5, R80 (2004)
12. Kimmig, A., Demoen, B., Raedt, L.D., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming 11, 235–262 (2011)
13. Murtagh, F.: Multidimensional Clustering Algorithms, COMPSTAT Lectures, vol. 4. Physica-Verlag, Wuerzburg (1985)
14. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, Previous number = SIDL-WP-1999-0120 (November 1999), `http://ilpubs.stanford.edu:8090/422/`
15. R Development Core Team. R: A Language and Environment for Statistical Computing. R Found. for Stat. Comp., Vienna, Austria (2011), `http://www.R-project.org/`
16. Sato, T., Kameya, Y.: Parameter learning of logic programs for symbolic-statistical modeling. Journal of AI Research 15, 391–454 (2001)
17. Shannon, P.: RCytoscape: Display and manipulate graphs in Cytoscape. R package (2011)
18. Srinivasan, A.: The Aleph Manual. University of Oxford (2004)
19. Wielemaker, J., Angelopoulos, N.: Syntactic integration of external languages in Prolog. In: ICLP Workshop on Logic-based methods in Programming Environments (WLPE 2012), Budapest, Hungary, pp. 40–50 (September 2012)
20. Wielemaker, J., Costa, V.S.: On the portability of Prolog applications. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 69–83. Springer, Heidelberg (2011)
21. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming 12(1-2), 67–96 (2012)
22. Zhou, N.-F.: The language features and architecture of B-Prolog. Theory and Practice of Logic Programming 12, 189–218 (2012)