Real 2.0: User's Guide

Integrative statistics with R

Nicos Angelopoulos
Stoics.org.uk, London, UK
http://stoics.org.uk/~nicos

Vitor Costa Santos
CRACS-INESC Porto LA, Universidade do Porto, Porto, Portugal
http://www.dcc.fc.up.pt/~vsc/

September 4, 2016

Contents

Abstract

Real is a Prolog library for integrative statistics with the R software. 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.

Chapter 1

Introduction

Real is a minimalistic yet powerful Prolog library that provides access to all aspects of the *R* (R Development Core Team, 2012) statistical environment.

This document is currently incomplete, the primary reference for Real (versions 2 and later) is (Angelopoulos, Abdallah, & Giamas, 2016), for earlier versions see (Angelopoulos et al., 2013).

In this document we primarily use examples from `examples/for_real.pl`. The user is strongly advised to look into this file for example code.

1.1 Installation

Real was originally designed, developed and tested on *YAP 6.3.1* (Costa, Rocha, & Damas, 2012) 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 (Wielemaker & Angelopoulos, 2012) to the *SWI* (Wielemaker, Schrijvers, Triska, & Lager, 2012) 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 (Wielemaker & Costa, 2011).

Since version 1.4 (2014 onwards) Real has been developed on *SWI* and in particularly was a driving force for the extension introduced in *SWI 7* (Wielemakers, 2014) and takes full advantage of the new syntax to provide a even tighter syntactic integration of *R* code in Prolog.

The library and examples presented here can be downloaded from our website (`http://stoics.org.uk/~nicos/sware/real/`),

To install and use the current stable version in SWI-Prolog simply do:

```
?- pack_install(real).
?- [library(real)].
```

Current versions of Yap Prolog include the library (although not necessarily the latest Real release). When using a downloaded binary, the library can be loaded as any other system library.

```
?- [library(real)].
```

To install from sources one needs to specify the `--with-R` configure option. This also works if replacing the supplied Yap directory `packages/real` with the new sources directory.

Chapter 2

Interface

Real enables the communication between the Prolog system and *R*. The *R* environment is loaded 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*. Real 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 deterministic statistical computations.

2.1 Access predicates

The *R* language uses `<-` as one of its two assignment operators. In order to be as close to possible to *R* syntax, Real uses `<-/1` and `<-/2` to channel the bulk of the

interactions between the two systems. 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, without getting any results back to Prolog. 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, Real 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 parsed from Prolog terms to strings and passed to R for native parsing.
- specific Prolog terms map to R data objects and efficiently passed between Prolog and R via their respective C language interface.

More concretely, there are 4 distinct calling modes for `<-`/2:

`+Rexpr`	$<-$	`+PLdata`	(M_1)
`+Rexpr`	$<-$	`+Rexpr`	(M_2)
`-PLvar`	$<-$	`+Rexpr`	(M_3)

with

`PLvar` a free Prolog variable

`PLdata` a Prolog term that can be interpreted as data

`Rexpr` a Prolog term that will be translated to an R expression

The modes are inferred from the supplied arguments. On the RHS, lists and `c`/n terms are interpreted as Prolog data while anything else, is assumed an R expression. On the LHS a free variable ($PLvar$) will be instantiated with the data from R (the RHS expression). A non variable LHS is translated as a term holding an R expression to which the RHS will be assigned to.

Examples:

```
% mode M_1

?- x <- c(1,2,3).
?- <- x.                    % prints x from within R
[1] 1 2 3

?- y <- [1,2,3].
?- <- y.
[1] 1 2 3

% mode M_2
```

```
?- z <- x + y.
?- <- z.
[1] 3 4 5

% mode M_3

?- X <- x.
X = [1, 2, 3].

?- T <- x + z.
T = [4, 6, 8].
```

The R expression in the LHS is not constrained, but for the call to be executed successful it should be an assignable expression. For instance:

```
?- a <- [1,2,3].
?- a[2] <- 4.
?- <- a
[1] 1 4 3
```

a negative example follows:

```
?- x <- 1, y <- 2.
?- x + y <- 2 + 3.
Error in x + y <- 2 + 3 : could not find function "+<-"
```

Note that it is entirely possible to use `<-/2` in the following mode,

```
-PLvar  <-  +PLdata
```

but this is not particularly useful:

```
?- X <- [1,2,3].
   X = [1, 2, 3].
```

2.1.1 Fast and furious

Modes M_1 and M_3 are about transferring data between Prolog and R efficiently via the respective C language interfaces. In these modes Real transfers data between the two systems by creating appropriate data structures and populating them via C code.

Basic Prolog data types along with lists and c/n compound terms are considered to be Prolog data (`PLdata`, above), while free variables are to be instantiated by data from R. In combination, these distinguish the data transfer modes. All items within Prolog lists are considered to be data, whereas c/n terms are interpreted as data if and only if they contain basic data items in all their arguments that can be cast to a single data type. Otherwise they are considered as R expressions. The expression mode, M_2, assigns the result of an R function call to an R object. Real provides a convenient syntax of R expressions as term structures. In M_2 the expressions are translated to R syntax and then passed to R (see Section 2.4).

In the following example of mode M_1, a list of 6 Prolog integers is passed to the R variable `v` and then their average value is passed to Prolog variable `Avg`.

```
?-  v <- [0,1,1,2,3,5],
    Avg <- mean(v).

Avg = 2.0.
```

2.2 Data representation in R

R recognises several types of objects:

- *Floating* point numbers, *integers*, *Boolean* and *ascii* values (character strings) provide the base types.
- *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 *pairlists*, which represent lists pairing a name to value.
- Within compound objects 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, however as of SWI-7 the default semantics are that text in double quotes are intepreted as native strings. In R character strings are of type *character* (here abbreviated to *char*). The full list of supported translations with data are summarised by the following rules:

```
Prolog          ---  R
integer         <->  integer
```

```
float          <->  double
atom           <->  string
string (SWI-7) <->  string
true/false     <->  logical
```

Within *R* expressions the following differences are seen:

```
atom           <->  R object
+ atom          ->  string
```

The file in sources location `examples/for\_real.pl` contains a number of instructional examples. We show here a couple of introductory examples and some that highlight the differences in translations within Prolog data and *R* expressions.

```
?- a <- [abc,def].
?- <- a.
[1] "abc" "def"

?- a <- ["abc","def"].
?- <- a.
[1] "abc" "def"

?- t <- paste( "abc", "def" ).
?- <- t.
[1] "abc def"

?- t <- paste( abc, def ).
Error in paste(abc, def) : object 'abc' not found
ERROR: R was unable to digest your statement, either syntax or existance error.

?- b <- [true,false,true,true].
?- <- b.
[1]  TRUE FALSE  TRUE  TRUE
```

There are two facts worth noting regarding translation from Prolog to R. Firstly, if there are expressions that prove difficult to express as Prolog terms, the user can always wrap the the expression in single quotes and use *R* syntax. So, the following two statements are equivelant but the second passes the *R* expression as an atom

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

The second point is that the user can inspect the generated *R* expression that is passed to *R* with *debug(real)*.

```
?- debug( real ).
?- x <- 'as.integer(c(1,2,3))'.
   % Sending to R: x <- as.integer(c(1,2,3))
```

The main compound types supported by the interface are symbols, pairlists 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.

2.3 Vectors, pairlists 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. As an example, the R variable `v`, defined by

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

is a vector of type integer 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. So passing back the R vector, instantiates the response variable to the list of the 3 numbers.

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

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).

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|...].
```

Other data types in R include expressions and functions.

2.4 Translation of *R* expressions

Prolog terms representing arbitrary R expressions are parsed and placed into strings that are subsequently passed from Prolog to R for native parsing and 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 Real 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.

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. Real 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)`.

Chapter 3

Syntax

3.1 Syntactic issues

There is valid *R* syntax which results 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, Real allows the use of the operator '`..`', e.g.:

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

  The library's original name (*r..eal*) was a word play on the '`..`' operator. However, as of *SWI-7* there is no need for the double period as the following

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

  is now recognised syntax.

- *R* allows matrix subscripts. In the style of BProlog (Zhou, 2012), Real uses the '`^`' operator. Recent changes in Prolog syntax mean that the usual subscript notation is now valid Prolog syntax. Both the following are valid:

  ```
  a^[2] => a[2]
  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`. Real uses `*` for this purpose:

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

Note that Real 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 3.1.1. 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
```

- Real used `(.)` to denote *R* functions with zero arity, however recent changes in Prolog syntax mean that we can do away with the dot. So both of the following are now valid,

```
dev..off(.) => dev.off()
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")
and
source(+'Atom') => source(+"Atom")
```

With *SWI-7* douple quotes are a new type (strings) which can be interpreted directly to *R*'s strings

- Some *R* operators cannot be represented by valid Prolog terms, so we introduced some mnemonic mappings (see Section 3.2 below):

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

Real	R	Description
$a[x]$	$a[x]$	index access
$a[*, 3]$	$a[, 3]$	missing array index
mod	%%	modulo
//	%\%	integer division
@ * @	% * %	matrix multiplication
@^@	% o %	outer product of arrays
@in@	% $\sim$ %	set/list membership
!=	\=	not equal operator

Table 3.1: Syntax translations between R and Real.

- the interface enables mapping of NA values within arithmetic vectors and matrices to $\$NaN$. When passing numeric data from Prolog to R in addition to $\$NaN$, the empty atom (‘’) is also translated to R's NA value.
- With recent versions of SWI users have access to infinty values. There are propagated preperly between the two systems

```
?- r <- 1 / 0.
?- <- r.
[1] Inf

?- R <- r.
R = 1.0Inf.

?- R <- r, t <- R.
R = 1.0Inf.

?- <- t.
[1] Inf
```

The majority of R operators can be used unquoted as they are defined as infix operators and present no issues. Finally, expressions that Real cannot translate can always be passed as Prolog atoms or strings. (Wielemaker & Angelopoulos, 2012) discuss some of these issues. Table 3.1 shows the correspondance of operators that need intepretation from Prolog to R as the R form is illegal Prolog syntax.

3.1.1 Component inspection

R's *pairlists* provide named and indexed access to their elements. `^[[]]` can be used to identify the ith or named element, and ‘$’ for the named element.

```
?-  a <- [x=1,y=0,z=3],
    <- a,
    A <- a[[1]],
    B <- a["y"]],
    C <- a$z,
    D <- a[[+y]].

$x
[1] 1

$y
[1] 0

$z
[1] 3

A = 1,
B = D, D = 0,
C = 3.
```

R's *S*4 objects, (Becker, Chambers, & Wilks, 1988), comprise slots which can be accessed with the '@' operator.

```
<- setClass(+"track", representation(x=+"numeric", y=+"numeric")),
myTrack <- new(+"track", x = -4:4, y = exp(-4:4)),
<- print( myTrack@x ),
Y <- myTrack@y,
write( y(Y) ), nl,
<- setClass(+"nest", representation(z=+"numeric", t=+"track")),
myNest <- new(+"nest", z=c(1,2,3) ),
myNest@t <- myTrack,
myNest@t@x <- Y+1,  % good ex. for hidden vars.
<- myNest.
```

```
[1] -4 -3 -2 -1  0  1  2  3  4
y([0.01831563888873418,0.049787068367863944,0.1353352832366127,...])
An object of class "nest"
Slot "z":
[1] 1 2 3

Slot "t":
An object of class "track"
Slot "x":
[1]  1.018316  1.049787  1.135335  1.367879  2.000000  3.71...
[8] 21.085537 55.598150
```

op.	prec.	assoc.	comment
	access		
<-	950	fx	R expressions
<-	950	yfx	bi-directional R link
<<-	950	yfx	
	component/slot extraction		
@	400	yfx	slot extraction (from formal object)
$	400	yfx	component extraction (vectors, matrices, arrays and lists)
	arithmetic		
~	600	xfy	
@*@	400	yfx	
@o@	400	yfx	
@~@	400	yfx	

Table 3.2: Operators defined in Real.

```
Slot "y":
[1]  0.01831564  0.04978707  0.13533528  0.36787944  1.0000...
[7]  7.38905610 20.08553692 54.59815003


Y = [0.01831563888873418, 0.049787068367863944, 0.135...]
```

3.2 Operators

Real defines a number of operators, which are show in Table 3.2. From the R help pages (`http://stat.ethz.ch/R-manual/R-devel/library/base/html/Syntax.html`) @ and $ "require names or string constants on the right hand side" so we define them as yfx and translate the RHS as identifier.

3.3 Associated predicates

Real was developed as academic open source software. It is therefore appreciated if you cite the papers related to Real when you publish work in which it played a beneficial role. To find relevant citations:

```
?- once(r_citation( Cit, bibtex(Type,Key,Entries) )),
   write( Cit ), nl, nl, member( E, Entries), write( E ), nl,
   fail.
```

```
Advances in integrative statistics for logic programming
Nicos Angelopoulos, Samer Abdallah and Georgios Giamas
International Journal of Approximate Reasoning,
http://dx.doi.org/10.1016/j.ijar.2016.06.008.

author=Nicos Angelopoulos, Samer Abdallah and Georgios Giamas
title=Advances in integrative statistics for logic programming
journal=Journal of Approximate Reasoning
year=2016
month=July
url=http://dx.doi.org/10.1016/j.ijar.2016.06.008
```

The current version and publication date can be found in `r_version`/3. The arguments hold the release number (Mj:Mn:Fx), the release date (`date`/3) and a codename or brief release note (atom).

```
?- r_version( A, B, C ).

A = 2:0:0,
B = date(2016, 8, 24),
C = ijar.
```

To debug the system and spy on the traffic between Prolog and *R* we use the *debug* SWI library (also available on YAP). The current support is not complete but can be instructive when the user gets unexpected behaviour. Cebugging is enabled and disabled as per normal `debug`/1 use:

off debugging/informational messages.

```
?-  debug(real) .
?-  nodebug(real).
```

Acknowledgements

We owe many thanks to Jan Wielemaker for re-drafting the whole of the *C* code in a single afternoon and evening. (Exclusive of dinner time at his local restaurant.) Also for tidy-ups to version 2.0 which were derived from his work in adjusting Real for Rserve. Thanks also to Samer Abdallah for work on the integration with web-services.

Index

References

Angelopoulos, N., Abdallah, S., & Giamas, G. (2016, July). Advances in integrative statistics for logic programming. *International Journal of Approximate Reasoning*. doi: http://dx.doi.org/10.1016/j.ijar.2016.06.008

Angelopoulos, N., Costa, V. S., Azevedo, J., Wielemaker, J., Camacho, R., & Wessels, L. (2013, Jan.). Integrative functional statistics in logic programming. In *Proc. of Practical Aspects of Declarative Languages* (Vol. 7752, p. 190-205). Rome, Italy.

Becker, R. A., Chambers, J. M., & Wilks, A. R. (1988). *The new S language: A programming environment for data analysis and graphics*. USA: Wadsworth & Brooks/Cole.

Costa, V. S., Rocha, R., & Damas, L. (2012). The YAP Prolog system. *Journal of Theory and Practice of Logic Programming*, *12*, 5-34.

R Development Core Team. (2012). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. (`http://www.R-project.org/`)

Wielemaker, J., & Angelopoulos, N. (2012, September). Syntactic integration of external languages in Prolog. In *ICLP workshop on logic-based methods in programming environments (WLPE'12)* (p. 40-50). Budapest, Hungary.

Wielemaker, J., & Costa, V. S. (2011). On the portability of Prolog applications. In *Practical Aspects of Declarative Languages (PADL'11)* (p. 69-83).

Wielemaker, J., Schrijvers, T., Triska, M., & Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, *12*(1-2), 67–96.

Wielemakers, J. (2014, July). SWI-Prolog version 7 extensions. In *ICLP workshop on logic-based methods in programming environments (WLPE'14)*.

Zhou, N.-F. (2012, January). The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, *12*, 189-218.