Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

rolog: Prolog queries from R

Matthias Gondan
Universität Innsbruck

Jan Wielemaker
SWI-Prolog Solutions b.v.

Abstract

Prolog is a classical logic programming language with many applications in expert systems, computer linguistics and traditional, that is, symbolic artificial intelligence. The main strength of Prolog is its concise representation of facts and rules for the representation of knowledge and grammar, as well as its very efficient built-in search engine for closed world domains. R is a statistical programming language for data analysis and statistical modeling which is widely used in academia and industry. Besides the core library, a lot of packages have been developed for all kinds of statistical problems, including new-style artificial intelligence tools such as neural networks for machine learning and deep learning. Whereas Prolog is weak in statistical computation, but strong in symbolic manipulation, the converse may be said for the R language. SWI-Prolog is a widely used Prolog system that offers a wide range of extensions for real world applications, and there already exist two Prolog "packs" to invoke R (**rserve-client**, **real**) from SWI-Prolog. However, given the large user community of R, there may also be a need for a connection in the reverse direction that allows invoking Prolog queries in R computations. The R package **rolog** embeds the SWI-Prolog system, thus enabling deterministic and non-deterministic queries to the Prolog interpreter. Usage of **rolog** is illustrated by a few examples.

Keywords: Statistics; Logic Programming; Artificial Intelligence, R, Prolog.

1. rolog: Prolog queries from R

The R (R Core Team 2021) programming language and environment is a widely used open source software for statistical data analysis. The basic R is a functional language with lots of support for storage and manipulation of different data types, and a strong emphasis on operations involving vectors and arrays. Moreover, a huge number of packages (e.g., CRAN, https://cran.r-project.org/) have been contributed that cover problems from diverse areas such as bioinformatics, machine learning, specialized statistical methods, web programming and connections to other programming languages. An interface to Prolog is lacking so far.

The logic programming language Prolog was invented in the 1970ies by Colmerauer and Roussel (1996), mostly for the purpose of natural language processing. Since then, logic programming has become an important driving force in research on artificial intelligence, natural language processing, program analysis, knowledge representation and theorem proving (Shoham 1994; Lally and Fodor 2011; Carro 2004; Hsiang and Srivas 1987). SWI-Prolog (Wielemaker, Schrijvers, Triska, and Lager 2012) is an open-source implementation of Prolog that mainly targets developers of applications, with many users in academia, research and industry. SWI-Prolog includes a large number of libraries for "the real world", for example, a web server, encryption, interfaces to C/C++ and other programming languages, as well as a development environment and debugger. In addition, pluggable extensions (so-called packs) are available for specific tasks to enhance its capabilities.

Unlike R, Prolog is a declarative programming language consisting of facts and rules that define relations, for example, in a problem space. Prolog's major strength is its built-in query-driven search engine that efficiently deals with complex structured data, with the data not necessarily being numerical. In fact, Prolog only provides a basic collection of arithmetic calculations via a purely functional interface `is/2`. More complex calculations such as matrix algebra, statistical models or machine learning need help from other systems, for example, from R.

Angelopoulos, Costa, Azevedo, Wielemaker, Camacho, and Wessels (2013) summarize work at the intersection of symbolic knowledge representation and statistical inference, especially in the area of model fits (Sato and Kameya 2013; Angelopoulos and Cussens 2008, EM algorithms, MCMC) and stochastic logic programs (Cussens 2000; Kimmig, Demoen, Raedt, Costa, and Rocha 2011). One of the major strengths of logic programming is handling constraints; and a number of systems for constraint satisfaction tools have been developed (constraint logic programming on booleans, finite domains, reals, and intervals) for that purpose (e.g., Frühwirth 1998; Triska 2018). Some constraint handlers exist in R (see the CRAN task view for optimization problems), but more of them would be available via a bridge between R and Prolog.

Earlier approaches to connect Prolog and R have been published as SWI-Prolog packs (real, rserve-client Angelopoulos *et al.* 2013; Wielemaker 2021b) and as modules for the YAP system (Azevedo 2011). Whereas **real** establishes a direct link to an embedded instance of R, **rserve-client** communicates with a local or remote R service (Urbanek 2021). The former approach emphasizes speed, the latter might be preferred from a security perspective, especially in systems such as SWISH (Wielemaker, Lager, and Riguzzi 2015) that accept only a set of sandboxed commands for Prolog, but do not impose restrictions on R. A common feature of the two packages is that they provide an interface for R calls from Prolog, but not the other way round, that is, querying Prolog from R is not possible, so far.

rolog is an attempt to fill this gap, and to offer the possibility to raise Prolog queries in R scripts, for example, to perform efficient symbolic computations, searches in complex graphs, parsing natural language and definite clause grammars. In addition, two Prolog predicates are provided that enable Prolog to ring back to the R system for bidirectional communication. Similar to **real**, tight communication between the two systems is established by linking to a shared library that embeds the current version of SWI-Prolog. The exchange of data is facilitated by the C++ interfaces of the two languages (Eddelbuettel and Balamuta 2018; Wielemaker 2021a). A less tight connection might be established using the recently developed machine query interface (Zinda 2021, MQI) that allows socket-based communication between foreign languages and SWI-Prolog (and, in fact, the **MQI** documentation includes an example in which R is called).

A bidirectional bridge between R and Prolog might overcome the limitations of both languages, thereby combining the extensive numerical and statistical power of the R system with Prolog's skills in the representation of knowledge and reasoning. In addition to the useful little tools shown in the examples below, **rolog** can therefore contribute to progress at the intersection of traditional artificial intelligence and contemporary statistical programming.

The next section presents the interface of **rolog** in detail. Section 3 presents possible extensions of the package at both ends, in R and Prolog. Section 4 is a list of illustrative examples that offer useful extensions to the R system. Conclusions and further perspectives are summarized in Section 5.

2. Basic syntax

rolog has a rather minimalistic syntax, providing only some basic ingredients to establish communication with an embedded SWI-Prolog. The ways to extend the interface are described in Section 3.

After installation (in R, with `install.packages("rolog")`), the package is loaded in the standard way using R's `library`-command.

```
R> library(rolog)
```

We can see SWI-Prolog's typical welcome message.

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.5.10-52-gcd28f88fd-DIRTY)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).
```

2.1. R interface

Most of the work can done using the three R functions `query`, `submit`, and `clear`. The functions `consult`, `once`, and `findall` are provided for convenience.

`consult`. In most applications, a number of Prolog facts and rules will be loaded into the system. To facilitate this recurrent task, the Prolog directive `consult/1` has been mirrored into R, `consult(filename)`, with *filename* given as a string (or a list of strings if multiple files are to be consulted). Note that the full filename should be given, including the extension (e.g., ".pl"). The function returns `TRUE` on success; in case of problems, it returns `FALSE` and an error message is shown.

`query`. The function `query(call, options)` is used to create a Prolog query (without invoking it yet). The first argument *call* is a regular R call that is created using R's function `call(name, ...)`. This call represents the Prolog predicate which will be queried in the later course. The creation of such predicates and Prolog terms is described below and can become quite contrived (see the examples in Section 4). The second argument, *options*, may be used for ad hoc modifications of the translation between R and Prolog, see the section below. The function returns `TRUE` on success. Note that the function does not check if the corresponding Prolog predicate actually exists (but see `submit()` below).

Only a single query can be opened at a given time. If a new query is created while another query is still open, a warning is shown and the other query is closed.

`submit`. Once a query has been created, it can be submitted using `submit()`. If the query fails, the return value is `FALSE`. If the query succeeds, a list of constraints is returned, with bindings for the variables that satisfy the query. Repeated calls to submit are possible, returning the different solutions of a query (until it eventually fails). Programmatically distinguishing between the different types of return values for success and failure (list vs. `FALSE`) is facilitated by the R function `isFALSE(x)`.

`clear`. Closes the query. The name of the function is chosen to avoid name clashes with R's own built-in function `close`. The function returns an invisible `TRUE`, even if there is no open query.

The R program in Listing 1 illustrates a query to Prolog's `member/2` using **rolog**'s syntax rules.

```
R> # member(1, [1, 2.0, a, "b", X])
R> # return value is TRUE (query successfully created), with an attribute
R> # that shows the prolog representation of the query
R> query(call("member",
+             1L, list(1L, 2.0, quote(a), "b", expression(X), TRUE)))

[1] TRUE
attr(,"query")
[1] "member(1, [1, 2.0, a, b, X, true])"

R> # returns an empty list, stating that member(1, [1 | _]) is satisfied
R> submit()

list()

R> # returns a list, stating that the query is also satisfied if X = 1
R> submit()

$X
[1] 1

R> # close the query
R> clear()
```

Listing 1: A query to Prolog's `member/2` predicate

`once` and `findall`. The function `once(call, options)` is a convenience function that acts as a shortcut for `query(call, options)`, `submit()`, and `clear()`. Similarly, `findall(call, options)` abbreviates the commands `query(call, options)`, repetition of `submit()` until

Table 1: Creating Prolog terms from R

R	Prolog	Note/Alternatives
`expression(X)`	Variable `X`	not necessarily uppercase in R
`as.symbol(abc)`	Atom `abc`	`as.name`, `quote`
`TRUE`, `FALSE`, `NULL`	Atoms `true`, `false`, `null`	
`"abc"`	String `"abc"`	
`3L`	Integer `3`	
`3`	Float `3.0`	
`call("term", 1L, 2L)`	Compound `term(1, 2)`	see below for `list()` and `c()`
`list(1L, 2L, 3L)`	List `[1, 2, 3]`	
`list(a=1, b=2, c=3)`	List `[a-1, b-2, c-3]`	
`c(1, 2, 3)`	`#(1.0, 2.0, 3.0)`	vectors of length > 1
`c(1L, 2L, 3L)` or `1:3`	`%(1, 2, 3)`	
`c("a", "b", "c")`	`$$("a", "b", "c")`	
`c(TRUE, FALSE, NA)`	`!(true, false, na)`	

failure, and `clear()`, returning a list collecting the the return value of the individual calls to submit.

2.2. Creating Prolog terms in R

Table 1 summarizes the rules for the translation from R to Prolog. Most rules work in both directions, but a few exceptions exist. For example, there is an empty atom in Prolog, but no empty symbol in R, so the empty atom is translated to a character string in R.

Moreover, R is mostly vectorized, lacking support for scalar entities (in R, scalar entities are treated as vectors of length 1. Conversely, Prolog does not natively support vectors or matrices. The problem is solved in the following way:

- R vectors of length 0 are translated to Prolog's empty list.
- R vectors of length 1 are translated to Prolog scalars.
- R vectors of length $N > 1$ are translated to Prolog terms `#/N`, `%/N`, `$$/N`, and `!/N` for floating point numbers, integers, strings and logicals, respectively.

In the reverse direction, Prolog terms like `#/N` are translated back to R vectors of length N, including terms `#/0` and `#/1` that map to R vectors of length 0 and 1, respectively. To summarize, the rules for translation are not fully symmetrical. A quick check for symmetry of the representation is obtained by a query to `r_eval/2` (see also below, subsection Prolog interface):

```
R> once(call("r_eval", c(1, 2, NA, NaN, Inf), expression(X)))

$X
[1]   1   2  NA NaN Inf
```

2.3. Package options

A few package-specific options have been defined to allow some fine-tuning of the rules for translation between R and Prolog.

- *realvec* (string): Name of the Prolog term for vectors of floats (default is `#`)
- *intvec* (string): same for vectors of integers (default is `%`)
- *boolvec* (string): same for vectors of logicals (default is `!`)
- *charvec* (string): same for vectors of character strings (default is `$$`). The single dollar cannot be used because it is the list operator in R.
- *scalar* (logical): if `TRUE` (default), R vectors of length 1 are translated to scalars in Prolog. If `FALSE`, R vectors are always translated to `#/N` etc., depending on the type.
- *portray* (logical): if `TRUE` (default in `query`), the result of `query`, `once` and `findall` includes an attribute with a representation of the query in Prolog.

The command `rolog_options()` returns a list with all the options. The options can be globally modified like this:

```
R> options(rolog.intvec="%%")
```

In a given query, the options can be set in the optional argument, for example,

```
R> query(call("member", expression(X), list(1:3, 4:6)),
+        options=list(intvec="%%"))
```

2.4. Prolog interface

rolog offers some basic support to call R from Prolog, that is, connecting the two systems in the reverse direction. Two predicates can be used for this purpose, `r_eval(Call)` and `r_eval(Function, Result)`. The former just invokes R with the command *Call* (ignoring the result); the latter evaluates *Function* and unifies the result with *Result*. Note that proper quoting of R functions is needed at the Prolog end, especially with R functions that start with uppercase letters or contain a dot in their name.

Two use cases for `r_eval/2` are shown in Section 4 below.

3. Extending the package

The package is intentionally kept minimalistic, but can easily be extended by convenience functions on both ends, Prolog and R, to facilitate recurrent tasks and/or avoid cumbersome syntax. R is a functional language, whereas Prolog is declarative. Obviously, there cannot be a perfect one-to-one correspondence between the syntactic components of two programming languages that follow completely different paradigms. Whereas symbols, functions, numbers and character strings are easily mapped between R and Prolog, there are loose ends at both sides. In particular, Prolog variables are translated from and to R *expressions* (not to be confused with R symbols), and R vectors of length $N > 1$ are translated to the Prolog terms `#/N`, `%/N`, `!/N`, and `$$/N`, as mentioned above. These rules are, in principle, arbitrary and can be intercepted at several stages. The process is illustrated in Figure 1.

- R functions that may be used to pre-process specific R elements before translation to Prolog (see, e.g., the R function `as.rolog()`)
- Prolog wrappers that manipulate the term before it is called and afterwards (see the example with dicts below)
- R functions that post-process the result of a query

3.1. Pre-processing in R

We have seen above that raising even simple everyday Prolog queries such as `member(X, [1, 2, 3, a, b])` require complicated R expressions such as `call("member", expression(X), list(1, 2, 3, quote(a), quote(b)))`. The R function `as.rolog(Call)` is meant to simplify this a bit by translating symbols starting with a dot to Prolog variables, and calls like `""[1, 2, 3, a, b]` to lists. The argument *Call* is typically a quoted R call or symbol:

```
R> q <- quote(member(.X, ""[1, 2, 3, a, b]))
R> as.rolog(q)

member(expression(X), list(1, 2, 3, a, b))
```

Note that the name of the variable will still be X in the later course, not "dot-X". A bit flexibility is lost because `quote()` treats the arguments `a`, `b` as symbols; to evaluate the respective variables (i.e., "unquote"), they can be put in parentheses:

```
R> a <- 4
R> b <- 5
R> q <- quote(member(.X, ""[1, 2, 3, a, (b)]))
R> as.rolog(q)

member(expression(X), list(1, 2, 3, a, 5))
```

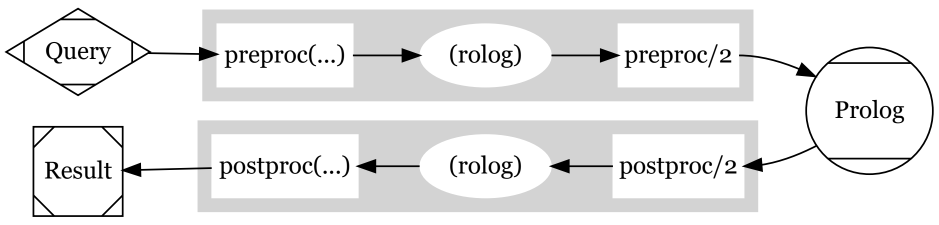

Figure 1: Workflow in **rolog**

Section 4 includes an example for mathematical rendering of R expressions. In that example, a pre-processing function is used to bring function calls with named arguments to a canonical form which is then handled in Prolog. More sophisticated work with quasi-quotations and unquoting expressions is described in "Advanced R" (Wickham 2019).

3.2. Post-processing in R

This may again be a function that reverts some of the manipulations during pre-processing. For `once()` and `submit()`, such a function would operate on the bindings. For example, many Prolog programmers are used to operate with atoms instead of character strings, whereas the latter is the preferred representation of symbolic information in R. The following simple example illustrates conversion of the results for a query like `member(X, [a, b, c])` to strings.

```
R> stringify <- function(x)
+  {
+    # replace Prolog variable by the value of the R variable
+    if(is.name(x))
+      return(as.character(x))
+
+    # Recurse into lists and calls
+    if(is.call(x))
+      x[-1] <- lapply(x[-1], FUN=stringify)
+
+    if(is.list(x))
+      x <- lapply(x, FUN=stringify)
+
+    # Leave the rest unchanged
+    return(x)
+  }
R> q <- quote(member(.X, ""[a, b, c]))
R> r <- findall(as.rolog(q))
R> stringify(r)
```

3.3. Pre- and post-processing in Prolog

Recent versions of SWI-Prolog support so-called dictionaries of the form `TagKey1:Value1, Key2:Value2, ...`. The tag is typically an atom (but can be a variable, as well), the keys are unique atom or integers; the values can be anything. Suppose we have a Prolog predicate that does something with dicts, and we would like to query it from R. The simplest solution is a wrapper in Prolog that translates *key-value* pairs `[Key1-Value1, Key2-Value2, ...]` back and forth to dicts:

```
do_something_with_pairs(Pairs0, Pairs1) :-
    dict_pairs(Dict0, my_dict, Pairs0),
    do_something_with_dicts(Dict0, Dict1),
    dict_pairs(Dict1, my_dict, Pairs1).
```

`do_something_with_pairs/2` can then be queried from R using, for example, lists with named elements (see Table 1).

```
R> once(call("do_something_with_pairs", list(a=1, b=2), expression(X)))
```

In the code above, `dict_pairs/2` takes the role of both `preproc/2` and `postproc/2` in Figure 1. It illustrates that complicated syntax on the R side can be much simplified when doing the conversion at the Prolog end. Ways to extend Prolog by add-ons ("packs") are shown in the next section.

4. Examples

In this section we present a few usage examples for the **rolog** package in increasing complexity. Although the code snippets are mostly self-explanatory, some familiarity with the Prolog language is helpful.

4.1. Hello, world

Prolog's typical *hello world* example is a search through a directed acyclic graph (DAG), for example, a family tree like the one given in Listing 2.

```
parent(pam, bob). parent(bob, ann). parent(bob, pat). parent(pat, jim).

ancestor(X, Z) :-
    parent(X, Z).

ancestor(X, Z) :-
    parent(X, Y),
    ancestor(Y, Z).
```

Listing 2: A family tree in Prolog (see also `family.pl` in the "pl" folder of the package)

Listing 2 is included in the package is accessed using the function `system.file(...)`. Within Prolog, the normal workflow is to consult the code with `[family]` and then to raise queries such as `ancestor(X, jim)`, which returns, one by one, four solutions for the variable *X*. In R, we obtain the following results:

```
R> library(rolog)
R> consult(system.file(file.path("pl", "family.pl"), package="rolog"))
R> query(call("ancestor", expression(X), quote(jim)))

[1] TRUE
attr(,"query")
[1] "ancestor(X, jim)"

R> submit()        # solutions for X
```

```
$X
pat

R> submit()        # etc.

$X
pam

R> clear()         # close the query
```

As stated above, `consult()` loads the facts and rules of Listing 2 into the Prolog database. `query(expr)` initializes the query *expr*, and the subsequent calls to `submit` return the conditions under which the query succeeds. In this example, the query succeeds if X is either `pat`, `pam`, or `bob`. A query is closed with `clear()`, or automatically if the query fails. Note that it is generally not possible to open two queries simultaneously, so opening a second query while another one is still open will raise a warning. If we are interested in just the first solution, we can use `once(expr)` as a shortcut to `query(expr)`, then `submit()`, then `clear()`. If we want to collect all solutions of a query with a finite set of solutions, we can use `findall(expr)`.

As mentioned in Section 3, a simplified syntax is provided by `as.rolog(...)` that accepts quoted expressions with dots indicating Prolog variables:

```
R> q <- quote(ancestor(.X, jim))
R> findall(as.rolog(q))
```

4.2. Backdoor test

A useful application of DAGs is confounder adjustment in causal analysis (Greenland, Pearl, and Robins 1999; Barrett 2021). The Prolog file `backdoor.pl` is an implementation of Greenland et al.'s criteria for the backdoor test for *d*-separation in DAGs, with a predicate `minimal/3` that searches for minimally sufficient sets of variables for confounder adjustment on the causal path between exposure and outcome.

```
R> consult(system.file(file.path("pl", "backdoor.pl"), package="rolog"))
R> # Figure 12 in Greenland et al.
R> add_node <- function(N)
+          invisible(once(call("assert", call("node", N))))
R> add_arrow <- function(X, Y)
+          invisible(once(call("assert", call("arrow", X, Y))))
R> add_node("a"); add_node("b"); add_node("c"); add_node("f"); add_node("u")
R> add_node("d") # outcome
R> add_node("e") # exposure
R> add_arrow("a", "d"); add_arrow("a", "f"); add_arrow("b", "d")
R> add_arrow("b", "f"); add_arrow("c", "d"); add_arrow("c", "f")
R> add_arrow("e", "d"); add_arrow("f", "e"); add_arrow("u", "a")
R> add_arrow("u", "b"); add_arrow("u", "c")
R> r <- findall(call("minimal", "e", "d", expression(S)))
R> unlist(r, recursive=FALSE)
```

```
$S
$S[[1]]
[1] "a"

$S[[2]]
[1] "b"

$S[[3]]
[1] "c"


$S
$S[[1]]
[1] "f"
```

The query to `minimal/3` returns two minimally sufficient sets of covariates for confounder adjustment (namely, {a, b, c} and {f}). The extra line with `unlist` only serves to tighten the output.

```
:- use_module(library(dcg/basics)).
s(s(NP, VP)) --> np(NP, C), blank, vp(VP, C).
np(NP, C) --> pn(NP, C).
np(np(Det, N), C) --> det(Det, C), blank, n(N, C).
np(np(Det, N, PP), C) --> det(Det, C), blank, n(N, C), blank, pp(PP).
vp(vp(V, NP), C) --> v(V, C), blank, np(NP, _).
vp(vp(V, NP, PP), C) --> v(V, C), blank, np(NP, _), blank, pp(PP).
pp(pp(P, NP)) --> p(P), blank, np(NP, _).
det(det(a), sg) --> `a`.
det(det(the), _) --> `the`.
pn(pn(john), sg) --> `john`.
n(n(man), sg) --> `man`.
n(n(men), pl) --> `men`.
n(n(telescope), sg) --> `telescope`.
v(v(sees), sg) --> `sees`.
v(v(see), pl) --> `see`.
p(p(with)) --> `with`.

% Translate R string to code points and invoke phrase/2
sentence(Tree, Sentence) :-
    string_codes(Sentence, Codes),
    phrase(s(Tree), Codes).
```

Listing 3: Simple grammar and lexicon. `sentence/2` pre-processes the R call

4.3. Definite clause grammars

One of the main driving forces of Prolog development was natural language processing (Dahl

1981). Therefore, the next example is an illustration of sentence parsing using so-called definite clause grammars. As Listing 3 shows, **rolog** can access modules from SWI's standard library (e.g., `"dcg/basics.pl"` below).

As in the first example, we first consult a little Prolog program with a minimalistic grammar and lexicon (Listing 3), and then raise a query asking for the syntactic structure of "john sees a man with a telescope". Closer inspection of the two results reveals the two possible meanings, "john sees a man *who carries* a telescope" versus "john sees a man *through* a telescope". More Prolog examples of natural language processing are found in Blackburn and Bos (2005), including the resolution of anaphoric references and the extraction of semantic meaning.

```
R> consult(system.file(file.path("pl", "telescope.pl"), package="rolog"))
R> findall(call("sentence",
+                expression(Tree), "john sees a man with a telescope"))
```

```
[[1]]
[[1]]$Tree
s(pn(john), vp(v(sees), np(det(a), n(man), pp(p(with), np(det(a),
    n(telescope))))))


[[2]]
[[2]]$Tree
s(pn(john), vp(v(sees), np(det(a), n(man)), pp(p(with), np(det(a),
    n(telescope)))))
```

4.4. Add-ons for Prolog

In description of the previous example, I noted in passing that **rolog** can access the built-in libraries of SWI-Prolog (e.g., by calls to `use_module/1,2`). It is also possible to extend the installation by add-ons, including add-ons that require compilation, if the build tools (essentially, RTools under Windows) are properly configured. This is illustrated below by the demo add-on **environ** (Wielemaker 2012) that collects the current environment variables.

```
R> once(call("pack_install",
+             quote(environ), list(call("interactive", FALSE))))
R> once(call("use_module", call("library", quote(environ))))
R> once(call("environ", expression(X)))
```

The query then unifies X with a list with `Key=Value` terms. The purpose if this example is obviously not to mimic the built-in function `Sys.getenv()` from R, but to illustrate the installation and usage of Prolog extensions from within R.[1]

[1]In most situations, the user would install the pack from within Prolog with `pack_install(environ)`.

4.5. Term manipulation

Prolog is homoiconic, that is, code is data. In this example, we make use of Prolog's ability to match expressions against given patterns and modify these expressions according to a few predefined "buggy rules" (Brown and Burton 1978), inspired by recurrent mistakes in the statistics exams of the first author's students. Consider the t-statistic for comparing an observed group average to a population mean:

$$T = \frac{\overline{X} - \mu}{s / \sqrt{N}} \tag{1}$$

Some mistakes may occur in this calculation, for example, omission of the implicit parentheses around the numerator and the denominator when typing the numbers into a calculator, resulting in $\overline{X} - \frac{\mu}{s} \div \sqrt{N}$, or forgetting the square root around N, or both. Prolog code for the two buggy rules is given in Listing 4.

```
% Correct step from task to solution
expert(tratio(X, Mu, S, N), frac(X - Mu, S / sqrt(N))).

% Mistakes
buggy(frac(X - Mu, S / SQRTN), X - frac(Mu, S) / SQRTN).
buggy(sqrt(N), N).

% Apply expert and buggy rules
step(X, Y) :-
    expert(X, Y).

step(X, Y) :-
    buggy(X, Y).

% Enter expressions
step(X, Y) :-
    compound(X),
    mapargs(search, X, Y),
    dif(X, Y).

% Search through problem space
search(X, X).

search(X, Z) :-
    step(X, Y),
    search(Y, Z).
```

Listing 4: Manipulating terms in Prolog

The little e-learning system shown in Listing 4 (`buggy.pl`) is invoked using the R script below.

```
R> consult(system.file(file.path("pl", "buggy.pl"), package="rolog"))
```

```
R> q <- quote(search(tratio(x, mu, s, n), .S))
R> s <- findall(as.rolog(q))
R> unlist(s)

$S
tratio(x, mu, s, n)

$S
frac(x - mu, s/sqrt(n))

$S
x - frac(mu, s)/sqrt(n)

$S
x - frac(mu, s)/n

$S
frac(x - mu, s/n)

$S
x - frac(mu, s)/n
```

The fourth and the sixth result are combinations of the two buggy rules (parenthesis, then square root, and the other way round). Some additional filters would be needed to eliminate trivial and redundant solutions (see, e.g., the chapter on generate-and-test in Sterling and Shapiro 1994).

It should be mentioned that R is homoiconic, too, and the Prolog code above can, in principle, be rewritten in R using non-standard evaluation techniques (Wickham 2019). Prolog's inbuilt pattern matching algorithm simplifies things a lot, though. An important feature of such a term manipulation is that the evaluation of the term can be postponed; for example, there is no need to instantiate the variables X, Mu, s, and N with given values before raising a query. This is especially helpful for variables that may represent larger sets of data in later steps.

4.6. Rendering mathematical expressions

The R extension of the markdown language (Xie, Dervieux, and Riederer 2020) enables reproducible statistical reports with nice typesetting in HTML, Microsoft Word, and Latex. However, so far, R expressions such as `pbinom(k, N, p)` are typeset as-is; prettier mathematical expressions such as $P_{\mathrm{Bi}}(X \le k; N, p)$ require Latex commands like `P_/mathrm{Bi}/left(X /le k; N, p/right)`, which are cumbersome to type in and hard to read even for simple equations. Since recently, manual pages include support for mathematical expressions (Sarkar and Hornik 2022), which already is a big improvement.

Below Prolog's grammar rules are used for an *automatic* translation of R expressions to MathML. The result can then be used for calculations or it can be rendered on a web page. A limited set of rules for translation from R to MathML is found in `pl/mathml.pl` of package **rolog**. The relevant code snippets are shown in the listings below, along with their output.

```
R> consult(system.file(file.path("pl", "mathml.pl"), package="rolog"))
R> mathml = function(term)
+  {
+    t = once(call("r2mathml", term, expression(X)))
+    cat(paste(t$X, collapse=""))
+  }
```

Listing 5: Using **rolog** to generate MathML from R expressions

The first example is easy. At the Prolog end, there is a handler for `pbinom/3` that translates the term into a pretty MathML syntax like $P_{\mathrm{Bi}}(X \leq k; N, \pi)$.

```
R> term = quote(pbinom(k, N, p))
R> mathml(term)     # pretty print

<math><mrow><msub><mi>P</mi><mtext>Bi</mtext></msub><mo>&af;</mo><mrow><mo>(</mo><mrow><mr

R> k = 10
R> N = 22
R> p = 0.4
R> eval(term)       # determine result

[1] 0.77195
```

The next example is interesting because Prolog needs find out the name of the integration variable for `sin`. For that purpose, **rolog** provides a predicate `r_eval/2` that calls R from Prolog (i.e., the reverse direction, see also next example). Here, the predicate is used for the R function `formalArgs(args(sin))`, which returns the name of the function argument of `sin`, that is, x.

```
R> term = quote(integrate(sin, 0L, 2L*pi))
R> mathml(term)

<math><mrow><munderover><mo>&int;</mo><mn>0</mn><mrow><mn>2</mn><mo>&#x2062;</mo><mi>&pi;<
```

Note that the Prolog end, the handler for `integrate/3` is quite rigid; it only accepts the three arguments *FUN*, *lower*, *upper* (in that particular order), and without names, that is, something like `integrate(sin, lower=0L, upper=2L * pi)` would not print the desired result. Moreover, `integrate` does not return a number, but a structured object with several slots.

```
R> canonical <- function(term)
+  {
+    if(is.call(term))
+    {
+      f <- match.fun(term[[1]])
+      if(!is.primitive(f))
+        term <- match.call(f, term)
+
+      # Recurse into arguments
+      term[-1] <- lapply(term[-1], canonical)
+    }
+
+    return(term)
+  }
R> # A custom function
R> g <- function(u)
+    sin(u)
R> # Mixture of (partially) named and positional arguments in unusual order
R> term <- quote(2L * integrate(low=-Inf, up=Inf, g)$value)
R> mathml(canonical(term))
```

```
<math><mrow><mn>2</mn><mo>&sdot;</mo><mrow><munderover><mo>&int;</mo><mrow><mo>-</mo><mi>&
```

Therefore, `pl/mathml.pl` includes two handlers. One handler accepts canonicalized terms with named arguments, `integrate(f=Fn, lower=Lower, upper=Upper)`. The extra function `canonical()` applies `match.call()` to non-primitive R calls, basically cleaning up the arguments and bringing them into the correct order. The other handler accepts terms of the form `$(integrate(Fn, Lower, Upper), value)` that are needed to access the numerical result of the integral.

4.7. Calling Prolog from R

The basic workflow of the bridge from R to Prolog is to (A) translate an R expression into a Prolog term (i.e., a predicate), (B) query the predicate, and then, (C) translate the result (i.e., the bindings of the variables) back to R. The reverse direction is straightforward, we start by translating a Prolog term to an R expression (i.e. Step C), evaluate the R expression, and then translate the result back to a Prolog term (Step A). **rolog** provides two predicates for this purpose, `r_eval(Expr)` and `r_eval(Expr, Res)`. The former is used to invoke an R expression *Expr* for its side effects (e.g., initializing a random number generator); it does not return a result. The latter is used to evaluate the R expression *Expr* and return the result *Res*. The code snippet in Listing 6 (`r_eval.pl`) illustrates this behavior.

```
r_seed(Seed) :-
    r_eval('set.seed'(Seed)).

r_norm(N, L) :-
    r_eval(rnorm(N), L).
```

Listing 6: Calling R from Prolog using `r_eval/1` and `r_eval/2`. The R call `set.seed` is quoted because the dot is an operator in Prolog.

```
R> consult(system.file(file.path("pl", "r_eval.pl"), package="rolog"))
R> once(call("r_seed", 1234))
R> once(call("r_norm", 3L, expression(X)))
```

The example in Listing 6 is a bit trivial, basically illustrating the syntax and the workflow. More serious applications of `r_eval/1,2` are illustrated in the example on mathematical expressions where `r_eval/2` is used to obtain the name of a function argument, as well as in the next example on interval arithmetic, where `r_eval/2` is used to evaluate monotonically behaving R functions.

4.8. Interval arithmetic

Let $\langle \ell, u \rangle$ denote a number between ℓ and u, $\ell \leq u$. It is easily verified that the result of the difference $\langle \ell_1, u_1 \rangle - \langle \ell_2, u_2 \rangle$ is somewhere in the interval $\langle \ell_1 - u_2, u_1 - \ell_2 \rangle$, and a number of rules exist for basic arithmetic operations and (piecewise) monotonically behaving functions (Hickey, Ju, and van Emden 2001). For ratios, denominators with mixed sign yield two possible intervals, for example, $\langle 1, 2 \rangle / \langle -3, 3 \rangle = \langle -\infty, 3 \rangle \cup \langle 3, \infty \rangle$, as shown in Figure 4 in Hickey et al.'s article. The number of possible candidates increases if more complicated functions are involved, as unions of intervals themselves appear as arguments (e.g., if $I_1 \cup I_2$ is added to $I_3 \cup I_4$, the result is $I_1 + I_3 \cup I_1 + I_4 \cup I_2 + I_3 \cup I_2 + I_4$). The bottom line is that calculations in interval arithmetic are non-deterministic in nature, and the number of possible results is not foreseeable and cannot, in general, be vectorized as is often done in R. Use cases for interval arithmetic are the control of limitations of floating-point representations in computer hardware, but intervals can also be used to represent the result of measurements with limited precision, or truncated intermediate results of students doing hand calculations. A few rules for basic interval arithmetic are found in `pl/interval.pl`; and examples are shown below. Again, in the example below, Prolog rings back to R via `r_eval/2` to determine the result of `dbinom(X, Size, Prob, Log)`.

```
R> consult(system.file(file.path("pl", "interval.pl"), package="rolog"))
R> q <- quote(int(`...`(1, 2) / `...`(-3, 3), .Res))
R> findall(as.rolog(q))

[[1]]
[[1]]$Res
...(-Inf, -0.333333333333333)



[[2]]
[[2]]$Res
...(0.333333333333333, Inf)

R> D  <- quote(`...`(5.7, 5.8))
R> mu <- 4
R> s  <- quote(`...`(3.8, 3.9))
R> N  <- 24L
R> tratio <- call("/", call("-", D, mu), call("/", s, call("sqrt", N)))
R> once(call("int", tratio, expression(Res)))
```

```
$Res
...(2.13545259627251, 2.32056923000512)

R> # Binomial density
R> prob = quote(`...`(0.2, 0.3))
R> once(call("int", call("dbinom", 4L, 10L, prob, FALSE), expression(Res)))

$Res
...(0.088080384, 0.200120949)
```

The slightly cumbersome syntax for entering an interval $\langle \ell, u \rangle$ is due to the fact that the ellipsis is a reserved symbol in R and cannot be used as an infix operator. A powerful and comprehensive system for constraint logic programming over intervals is available as a Prolog pack (Workman 2021) and can easily be connected to R using, for example, the present package.

5. Conclusions

R has become the primary language for statistical programming and data science, but is currently lacking support for traditional, symbolic artificial intelligence. There are already two add-ons for SWI-Prolog that allow to run R calculations from Prolog (Angelopoulos *et al.* 2013; Wielemaker 2021b), but a connection in the other direction was missing, so far. **rolog** bridges this gap by providing an interface to a SWI-Prolog distribution embedded into an R package. The communication between the two systems is mainly in the form of queries from R to Prolog, but two predicates allow Prolog to ring back and evaluate terms in R. The design of the package is minimalistic, providing three main functions `query(...)`, `submit()`, and `clear()`, and a very limited set of convenience tools (`consult(...)`, `once(...)`, and `findall(...)`) to facilitate recurrent everyday actions. As both systems are homoiconic in nature, it was easy to establish a one-to-one correspondence between many the elements of the two languages. Most exceptions (e.g., lack of R support for empty symbols) can be avoided and/or circumvented by wrapper functions at both ends.

Simple ways to extend the package have been described in Section 3; such extensions could, for example, include R objects and structures like those returned by `lm`, or S4 classes. In many use cases, this may be realized by transforming the R object to a list with named elements, and rebuild the object on the Prolog end on an as-needed basis. After a query, the process is reversed. If speed is an issue, more of these steps can, in principle, be moved into the package and implemented in **Rcpp**.

rolog, thus, opens up a wide of applications in logic programming for statisticians and researchers at the intersection of symbolic and connectionist artificial intelligence, where concise knowledge representation is combined with statistical power. Moreover, **rolog** provides starting points for useful small-scale solutions for everyday issues in data science (term transformations, pretty mathematical output, interval arithmetic, see Section 4).

At its present stage, a major limitation of **rolog** is its relatively slow speed. For example, translation of R lists or vectors to the respective elements of the Prolog language (also lists, `#/N`) is done element-wise, in both directions. The translation is optimized by using **Rcpp** (Eddelbuettel and Balamuta 2018), but there remains an upper bound in the efficiency, because

Prolog does not support vectors or matrices. Since Prolog's primary purpose is not vector or matrix calculation, this limitation may not show up in real-world applications. Another issue, maybe a bit annoying, is the rather cumbersome syntax of the interface, with the need for quoted calls and R expressions being a bit misused for representing Prolog variables. **rolog** was deliberately chosen to be minimalistic and, so far, only depends on base R. A more concise representation might be obtained by tools from the "Tidyverse" ecosystem, as described in Chapter 19 of Advanced R (Wickham 2019). Finally, at this stage, **rolog** is unable to deal with cyclic terms (e.g., `once(call("=", expression(A), call("f", expression(A))))` raises an error message).

rolog is available for R Version 4.2 and later, and can easily be installed using the usual `install.packages("rolog")`. The source code of the package is found at `https://github.com/mgondan/rolog/`, including installation instructions for Unix, Windows and macOS.

Acknowledgment

Development of the package profited substantially from the Prolog packs **rserve_client** (Wielemaker 2021b) and **real** (Angelopoulos *et al.* 2013). The project was financially supported by the European Commission (Erasmus+ project QHelp, 2019-1-EE01-KA203-051708).

Note

The results in this paper were obtained using R 4.2.0 with the **rolog** 0.9.4 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/`.

References

Angelopoulos N, Costa VS, Azevedo J, Wielemaker J, Camacho R, Wessels L (2013). "Integrative functional statistics in logic programming." In *Proceedings of Practical Aspects of Declarative Languages*, volume 7752 of *LNCS*, pp. 190–205. Rome, Italy.

Angelopoulos N, Cussens J (2008). "Bayesian learning of Bayesian networks with informative priors." *Journal of Annals of Mathematics and Artificial Intelligence*, **54**, 53–98.

Azevedo J (2011). *YapR*. URL `https://github.com/jcazevedo/YapR`.

Barrett M (2021). *ggdag: Analyze and Create Elegant Directed Acyclic Graphs*. R package version 0.2.4, URL `https://CRAN.R-project.org/package=ggdag`.

Blackburn P, Bos J (2005). *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI, Stanford.

Brown JS, Burton RR (1978). "Diagnostic models for procedural bugs in basic mathematical skills." *Cognitive Science*, **2**, 155–192.

Carro M (2004). *An Application of Rational Trees in a Logic Programming Interpreter for a Procedural Language*. Technical University of Madrid.

Colmerauer A, Roussel P (1996). "The Birth of Prolog." In TJ Bergin Jr, RG Gibson Jr (eds.), *History of Programming languages—II*, pp. 331–367. ACM, New York, NY, USA.

Cussens J (2000). "Stochastic logic programs: Sampling, inference and applications." In TJ Bergin Jr, RG Gibson Jr (eds.), *Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI00)*, p. 115122. Morgan Kaufmann Publishers, San Francisco, CA.

Dahl V (1981). "Translating Spanish into logic through logic." *American Journal of Computational Linguistics*, **7**(3), 149–164.

Eddelbuettel D, Balamuta JJ (2018). "Extending R with C++: A Brief Introduction to Rcpp." *The American Statistician*, **72**(1), 28–36. doi:10.1080/00031305.2017.1375990. URL https://doi.org/10.1080/00031305.2017.1375990.

Frühwirth T (1998). "Theory and Practice of Constraint Handling Rules." *Journal of Logic Programming*, **37**, 95–138.

Greenland S, Pearl J, Robins JM (1999). "Causal diagrams for epidemiologic research." *Epidemiology*, **10**, 37–48.

Hickey T, Ju Q, van Emden MH (2001). "Interval arithmetic: from principles to implementation." *Journal of the ACM*, **48**, 1038–1068.

Hsiang J, Srivas M (1987). "Automatic inductive theorem proving using prolog." *Theoretical Computer Science*, **54**(1), 3–28.

Kimmig A, Demoen B, Raedt LD, Costa VS, Rocha R (2011). "On the implementation of the probabilistic logic programming language ProbLog." *Theory and Practice of Logic Programming*, **11**, 235–326.

Lally A, Fodor P (2011). "Natural Language Processing With Prolog in the IBM Watson System."

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Sarkar D, Hornik K (2022). *Enhancements to HTML Documentation*. URL https://blog.r-project.org/2022/04/08/enhancements-to-html-documentation/index.html.

Sato T, Kameya Y (2013). "Parameter learning of logic programs for symbolic statistical modeling." *Journal of AI Research*, **15**, 391454.

Shoham Y (1994). *Artificial intelligence techniques in prolog*. Morgan Kaufman, San Francisco.

Sterling L, Shapiro E (1994). *The Art of Prolog*. MIT Press, Cambridge.

Triska M (2018). "Boolean constraints in SWI-Prolog: A comprehensive system description." *Science of Computer Programming*, **164**, 98–115. ISSN 0167-6423. Special issue of selected papers from FLOPS 2016.

Urbanek S (2021). *Rserve: Binary R server*. R package version 1.8-10, URL https://CRAN.R-project.org/package=Rserve.

Wickham H (2019). *Advanced R*. Chapman and Hall/CRC, Cambridge.

Wielemaker J (2012). *Demo package with C code, fetching the program environment*. URL https://www.swi-prolog.org/pack/list?p=environ.

Wielemaker J (2021a). *A C++ interface to SWI-Prolog*. URL https://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/pl2cpp.html%27).

Wielemaker J (2021b). *Rserve client for SWI-Prolog/SWISH*. URL https://github.com/JanWielemaker/rserve_client.

Wielemaker J, Lager T, Riguzzi F (2015). "SWISH: SWI-Prolog for Sharing." *CoRR*, **abs/1511.00915**. 1511.00915, URL https://arxiv.org/abs/1511.00915.

Wielemaker J, Schrijvers T, Triska M, Lager T (2012). "SWI-Prolog." *Theory and Practice of Logic Programming*, **12**(1-2), 6796. doi:10.1017/S1471068411000494.

Workman R (2021). *clpBNR. Von CLP over reals using interval arithmetic. Includes rational, integer and Boolean domains as subsets*. URL https://github.com/ridgeworks/clpBNR.

Xie Y, Dervieux C, Riederer E (2020). *R Markdown Cookbook*. Chapman and Hall/CRC, Cambridge.

Zinda E (2021). *MQI – Python and Other Programming Languge Integration for SWI Prolog*. URL https://www.swi-prolog.org/pldoc/man?section=mqi-overview.

Affiliation:

Matthias Gondan
Department of Psychology
University of Innsbruck
Innrain 9
A-6020 Innsbruck
E-mail: Matthias.Gondan-Rochon@uibk.ac.at

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd