Advances in integrative statistics for logic programming

Nicos Angelopoulos[1,e], Samer Abdallah[f], Georgios Giamas[e]

[a]*Welcome Trust Sanger Institute, Hinxton, CB10 1SA, UK*
[b]*Department of Surgery and Cancer, Imperial College, London, UK*
[c]*Department of Computer Science, University College London, UK*

We present recent developments on the syntax of *Real*, a library for interfacing two Prolog systems to the statistical language *R*. We focus on the changes in Prolog syntax within SWI-Prolog that accommodate greater syntactic integration, enhanced user experience and improved features for web-services. We recount the full syntax and functionality of *Real* as well as presenting a full application and sister packages which include Prolog code interfacing a number of common and useful tasks that can be delegated to *R*. We argue that *Real* is a powerful extension to logic programming, providing access to a popular statistical system that has complementary strengths in areas such as machine learning, statistical inference and visualisation. Furthermore, *Real* has a central role to play in the uptake of semantic web, computational biology and bioinformatics as application areas for research in logic programming.

Email address: `nicos.angelopoulos@sanger.ac.uk` (Nicos Angelopoulos)

Advances in integrative statistics for logic programming

Nicos Angelopoulos[1,e], Samer Abdallah[f], Georgios Giamas[e]

[d]Welcome Trust Sanger Institute, Hinxton, CB10 1SA, UK
[e]Department of Surgery and Cancer, Imperial College, London, UK
[f]Department of Computer Science, University College London, UK

1. Introduction

Real [4] is a low level interface between Prolog and *R* [14]. It enables the user to call *R* functions on Prolog data and communicate the results back to the logic system. The library works on two open source systems: YAP [6] and SWI-Prolog [25]. This is possible as YAP has a fairly complete emulation of SWI's *C* language interface [23]. Since its first introduction *Real* has evolved and has exerted some influence in advances to Prolog syntax. Furthermore, it has been used in a number of projects and in the process acquired a number of sister libraries. These libraries deliver Prolog predicates to useful tasks that can be best be dealt by existing *R* code. *Real* has thus be shown to be a useful and well integrated Prolog library that can provide access to the wealth of open source code available in *R*.

Here we focus on describing the full syntax of *Real 2* and its role in recent developments with syntactic changes in *SWI-7*. The changes in both systems have made the integration of *R* code into Prolog more natural and unobtrusive. Changes in the library itself had to be made to accommodate transition to the new Prolog syntax while preserving compatibility with traditional implementations. It is thus the case that the *Real* can be used in both of the supported Prolog systems, but only SWI-Prolog benefits from the new tighter integration.

R has a huge array of contributed code often accompanying published papers. It has particular strengths in statistical inference [19, 8], machine learning [11, 9] and data visualisation [20]. Within the specialist area of bioinformatics, Bioconductor [7] is a large agglomerating project that manages a large number of additional, user-contributed libraries.

Real gives access to *R* libraries that can complement Prolog's weaknesses in areas such as statistical inference and visualisation. With the library installed, it is straight forward with a basic grasp of *R* to call its functions on Prolog data. However, for users with no prior exposure to *R* there still might be a barrier. To address this, and in order to increase general usability of the library a number

Email address: `nicos.angelopoulos@sanger.ac.uk` (Nicos Angelopoulos) of sister packages have been developed. We highlight some of the predicates that enable access to *R* code without any knowledge of *R*.

Central application areas since the inception of *Real* has been these of semantic web, bioinformatics and computational biology. In this paper we describe the role of *Real* in a web-based application as well as presenting sister libraries that here have evolved for addressing real world bioinformatics tasks in the context of a variety of projects: [27, 12, 16]. The main thesis of this paper is that Prolog can play a central role as a unifying platform in research in statistical and probabilistic areas such as web reasoning and bioinformatics, taking advantage of its strong grip on knowledge representation and reasoning and in combinations with recent advances with *Real* and web programming [24, 10].

2. ***Real***

In this section we describe the main features of *Real* and the innovations in the new version *Real 2*, which include: syntactic extensions that allow *R* code to be represented in a form that more closely resembles normal *R* syntax, the new predicate *r_library*/1, which provides a more flexible way to locate and load *R* libraries from their local filestore, and support for multiple Prolog threads to use a single *R* session, allowing *Real* to be used in SWI-Prolog's multithreaded web server framework. Taken together, these innovations allow a tighter and smoother integration of *R* code and enable Prolog programmers to tap in the wealth of statistical functions implemented in *R* with greater ease.

2.1. Real's predicates

Real 2 adopts the convention of a uniform prefix to all the library predicates. The full list of *Real*'s predicates along with the associated operators and brief descriptions is shown in Table 1. New additions include a hookable locator for *R* libraries, web server support, intuitive syntax for non-destructive assignment and a new interface predicate for mixing Prolog and *R* options with options for directing output to graphical devices.

With the new predicate *r_library*/1 users can load the standard *R* libraries in their local installation. In addition, the predicate can be directed to user specified locations where local, possibly, changed sources of such libraries can be preferentially loaded in *Real*. The flexibility allows for (a) specific code to be loaded only known to *Real* thus leaving the remainder of the *R* installation intact, and (b) user code that can be made available and can work either with the distributed version while having extra functionality when used with the altered sources.

2.2. Basic operation

The bulk of the interaction with *Real* is via a single predicate $\leftarrow/2$ which is also defined as an infix operator. It is similar to the Prolog *is*/2 operator, except that the term on the right-hand side is interpreted as an *R* expression and evaluated in the embedded *R* session. Within *Real*, $\leftarrow/2$ can be used

Indicator	Operator	Symbol	Description
$r/2$	`<-`	$\leftarrow$	evaluate R expression and assign result
$r/1$	`<-`	$\leftarrow$	evaluate R expression and ignore result
$r\_new/1$	`<<-`	$\twoheadleftarrow$	argument is a fresh R variable
$\twoheadleftarrow /2$	`<<-`	$\twoheadleftarrow$	$r/2$ but with error if R variable exists
$r\_call/2$	`<-C++O`	$\leftarrow +\!\!+$	$r/\{1,2\}$ with options (O)
$r\_library/1$			load R library in a hookable manner
$r\_start/0$			start the connection to R
$r\_stop/0$			stop the connection to R
$r\_remove/1$			remove R variable
$r\_thread\_loop/0$			start an R thread server
$r\_serve/0$			serve all R expressions on queue thread

Table 1: Library's main predicates

to transfer data between R and Prolog, to apply R functions to Prolog data, retrieve R values as Prolog data, and destructively assigning values to R variables. Disambiguation clearly distinguishes the different modes, which can be summarised by:

$$-PlVar \leftarrow +Rexpr$$
$$+RAexpr \leftarrow +PlData$$
$$+RAexpr \leftarrow +Rexpr$$

Disambiguation of the call modes depends on whether the right hand side (RHS) is ground and if so on its term form. When the left hand side (LHS) of the operator is a free variable, the first mode is assumed, where the value of *Rexpr* is passed to *PlVar* after it has been evaluated in R. When the RHS is a c/N term or a list then the second mode is assumed and the Prolog data term in the RHS, *PlData*, is transferred to the assignable R expression in the LHS: *RAexpr*, typically an R variable. If none of the top two rules applies, *Real* passes the second argument to R for evaluation and the value assigned to the LHS term (RAexpr).

The following examples show how to: transfer Prolog data to R and back (1), transfer Prolog data to R and get the result of applying a function to the data in the new R variable (2) and demonstrating how to apply an R function on Prolog data without the use of an explicit R variable (3).

$$\begin{array}{l} ?-\ a \leftarrow [1,2,3],\quad A \leftarrow a. \\ A\ = [1,2,3]. \end{array} \tag{1}$$

$$\begin{array}{l} ?-\ a \leftarrow [1,2,3],\quad Mean \leftarrow mean(a). \\ Mean\ = 2.0. \end{array} \tag{2}$$

$$\begin{array}{l} ?-\ Mean \leftarrow mean([1,2,3]). \\ Mean\ = 2.0. \end{array} \tag{3}$$

On occasions the return value of an *R* expression is of no consequence. In such cases it can be ignored by using the single argument operator

$\leftarrow +Rexpr$

The following two examples illustrate its use. The first example plots Prolog data to the current graphic display, while the second example prints the contents of an *R* variable (x) to the output stream.

```
<- plot([1,2,2,4]).
<- x.
```

2.3. Composite data objects

As already seen, *Real* maps an *R* vector to a Prolog list of objects. In addition named lists are mapped to '='-pair lists and matrices to either nested lists or lists of compound terms. The following two examples pass a matrix and a named list to *R* variables x and y.

```
?- x <- [row(1,2,3),row(4,5,6)].
% same as:
?- x<-  [[1,2,3],[4,5,6]].
?- <- x.
       [,1]  [,2]  [,3]
 [1,]  1     2     3
 [2,]  4     5     6

?- X <- x.
X = [[1, 2, 3],  [4,  5,  6]].

?- y <- [a=[1,2,3],b=[4,5,6]].
?- <- y.
$a
[1]  1 2 3
$b
[1]  4 5 6
```

Through matrices and named lists, Prolog can also access data frames which are widely used in *R*. For instance, the values of the example data frame *mtcars* can be passed to Prolog as a matrix represented by a list of lists.

```
 ?- Mt <- as.matrix( mtcars ).
 Mt = [ [21.0,  6.0,  160.0,  110.0,  3.9,  2.62,  16.46,  0.0|...],
        [21.0,  6.0,  160.0,  110.0,  3.9,  2.875,  17.02|...],
        [22.8,  4.0,  108.0,  93.0,  3.85,  2.32|...],     ...   ].
```

2.4. Session variable management

In *R*, assigning the result of a computation to a variable, say *x*, has the side-effect of changing the state of the *R* session, *viz.*, binding the name 'x' to a new value, overwriting any binding it may have had previously. To avoid the risk of losing data in this way, we introduced operators $\Leftarrow/2$ and $\twoheadleftarrow/1$ and predicate *r_new*/1, provide fresh unused variable names on demand. The first ensures that its first argument (an *R* variable) did not exist prior to assigning to it some new values. The second removes its arguments from the *R* work-space and the third fails if its argument is already a known *R* variable.

2.5. Syntax

Although Prolog's hierarchical terms are sufficient for expressing the syntax of any *R* expression, the resulting calls to *R* would not be recognisable as *R* code by programmers. *Real* enhances the representation of *R* expressions by Prolog operators in order to be as close as possible to *R* syntax. In previous versions of *Real*, four aspects of *R* syntax where impossible to emulate in Prolog: the use of ('.') in *R* identifiers, the use of double quotes ('"') to represent strings, the invocation of functions with no arguments ('*foo*()'), and accessing array elements with subscripts in square brackets ('*a*[1,2]'). In these cases, Prolog operators were used to provide representations of such *R* syntactic constructions:

- Operator '..' was used to construct arity 2 terms to represent *R* identifiers containing a period. For example, the Prolog term *my..variable* was translated to the *R* identifier *my.variable*, *my..variable* $\longrightarrow$ *my.variable*).
- Operator + applied to a non numerical value resulted in a conversion from atoms and code lists to strings, +*foo* $\longrightarrow$ "foo".
- With the newly, at the time, introduced block operator '()' it was possible to represent the *R* syntax *foo*() as the Prolog term *foo*'()', *foo*() $\longrightarrow$ *foo*'()'.
- The infix operator '^' was used to represent array sub-scripting with a list as the second argument, i.e., *a*^[1,2] $\longrightarrow$ *a*[1,2]

With *Real* in mind, *SWI-7* [21] introduced extensions to Prolog syntax that allowed all of the above *R* syntactic constructs to be parsed in Prolog, under the control of per-module Prolog flags, such as *allow_dot_in_atom* (default *false*) and *double_quotes* (default *string*). *Real* has been adapted to utilise the new changes in a backwards compatible manner. Under YAP, *Real* continues to work using standard Prolog operator definitions and the old term representation, while under SWI-Prolog, users are able to take advantage of the more direct syntax enabled by the new extensions. In addition, the complexity of expressions such as those allowed in array indices has been enhanced. All of the following are now valid *Real* syntax mapping to the corresponding *R* constructs, provided the relevant flags have been enabled.

R	*Real*	Description
a[,3]	*a*ˆ[∗,3]	missing array index
%*%	@∗@	matrix multiplication
%o%	@ˆ@	outer product of arrays
%˜%	@*in*@	set/list membership
!=	\=	not equal operator

Table 2: Syntax translations between *R* and *Real*.

- *func.foo*(*a*,*b*,*c*)
- *write.csv*(”to_file.csv”, *x*)
- *foo*()
- *a*[1,2]
- *hmrn.pass* ← *hmrn*[*hmrn$*’Contamination’==”PASS”, ∗]

A few remaining translations, shown in Table 2 , handle *R* syntax that cannot be parsed directly as Prolog despite the recent syntax extensions. In *R*, a missing subscript denotes array *slicing*; e.g., *a*[,3] is the vector obtaining by extracting the third column of *a*. In *Real*, the missing subscript is represented by an asterisk, e.g, *a*[∗,3]. Similarly, certain combinations of symbols cannot be used to construct Prolog operators. *R* operators mapping to these combination must be represented using variabnt Prolog operators. The mapping of such operator mappings in *Real* is shown in Table 2.

A final innovation at the syntactic level has been the introduction of ‘NA’ values in the interface. In *R*, NA values stand for not available or unknown value placeholders. Prolog does not support such values internally, but the interface enables mapping of such 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.

In summary, there are only very few instances where *Real* code diverges syntactically from *R* code. In the vast majority of cases *Real* code is valid *R* code and it is well signposted with Prolog via the use of single main operator and call: ← /2.

2.6. Handling optional named arguments

Integral to the *R* language design and practice is the use of options that control the details of function calls. These options are pairs of argument name to values (*name=value*), which might not be necessarily present at invocation. When not present, default values supplied by the function developers are used. Similarly but not as widely used, is the use of list of terms that control calls to Prolog predicates. By convention an options list is placed at the last argument of a predicate and commonly contains a number of single arity terms. *Real 2* provides a uniform way of marrying the two conventions and a flexible way of handling options addressed to Prolog predicates accessing *R* functions. In addition, a number of standard tasks have been incorporated to a new interface predicate:

r_call(*Func*,*Opts*)

which can also be accessed as

$\leftarrow$ *Func* ++ *Opts*

Func is a compound term which is translated to an *R* function call and *Opts* can be a combination of: (a) =/2 terms, which are added to *Func*, (b) options controlling *r_call*/2's own execution and (c) Prolog style options which can influence the caller's behaviour but are ignored in the *R* call. The options term *Opts* is in general a list, but for convenience a singleton option list can be given as an non-list term. The main options for *r_call*/2 are:

rvar(*Rvar*)	: when given call becomes: *Rvar* $\leftarrow$ *Fcall*
rmv(*Rmv*= *false*)	: removes *Rvar* after end of call
stem(*Stem*= *real_plot*)	: stem to use for output files
outputs(*Outs*= *false*)	: a list of output devices
debug(*Dbg*= *false*)	: sets debug(real) for the duration of call
fcall(*Fcall*)	: returns the function call as presented to *R*
post_call(*Post*)	: call this after the function call

The following three calls illustrate the use of the new predicate. The first example enables debug messaging during the specific *R* call, the second redirects output of its plot call to a PDF file and the third passes an arbitrary argument to the *R* call.

$\leftarrow$ *plot*([1,2,3])++*debug*(*true*).
$\leftarrow$ *plot*([1,2,3])++*outputs*(*pdf*).
$\leftarrow$ *plot*([1,2,3])++*xlab*= "bespoke−x−label".

2.7. Multithreading and performance

R is inherently single threaded. To support the use of *Real* in multithreaded applications, and particularly in web servers built with SWI Prolog's HTTP libraries [24], *Real 2* allows a single designated *Real* server thread to be started. This thread controls evaluation of all *R* expressions. Subsequent calls to $\leftarrow$/1 and $\leftarrow$/2 from any thread are redirected to the *Real* server thread and the results awaited. Communication is handled synchronously using SWI Prolog queues and is transparent to the user.

The core of *Real* is implemented in *C,* providing an efficient and stable bridge which has been used to pass extremely large datasets to and from *R*. In comparison to the computational power required to compute with such large datasets the crossover time is negligible. In the following example predicate *numlist*/3 constructs the list of all integers appearing between its first and second arguments. *Real* can pass a million integers in a fraction of a tenth of a second on a modest computer (CPU at 2.30GHz) with its performance scaling linearly.

```
?- numlist(1,1000000,Million),time(mv ← Million),length(Million,Length).
% 30 inferences, 0.033 CPU in 0.033 sec (100% CPU, 483 Lips)
Million = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
Length = 1000000.


?- numlist(1,10000000,TenMillion), time(mv ← TenMillion).
% 118 inferences, 0.341 CPU in 0.341 sec (100% CPU, 346 Lips)
TenMillion = [1, 2, 3, 4, 5, 6, 7, 8, 9|...].

?- numlist(1,100000000,HndMillion), time(mv ← HndMillion).
% 116 inferences, 3.426 CPU in 3.427 sec (100% CPU, 34 Lips)
HndMillion = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
```

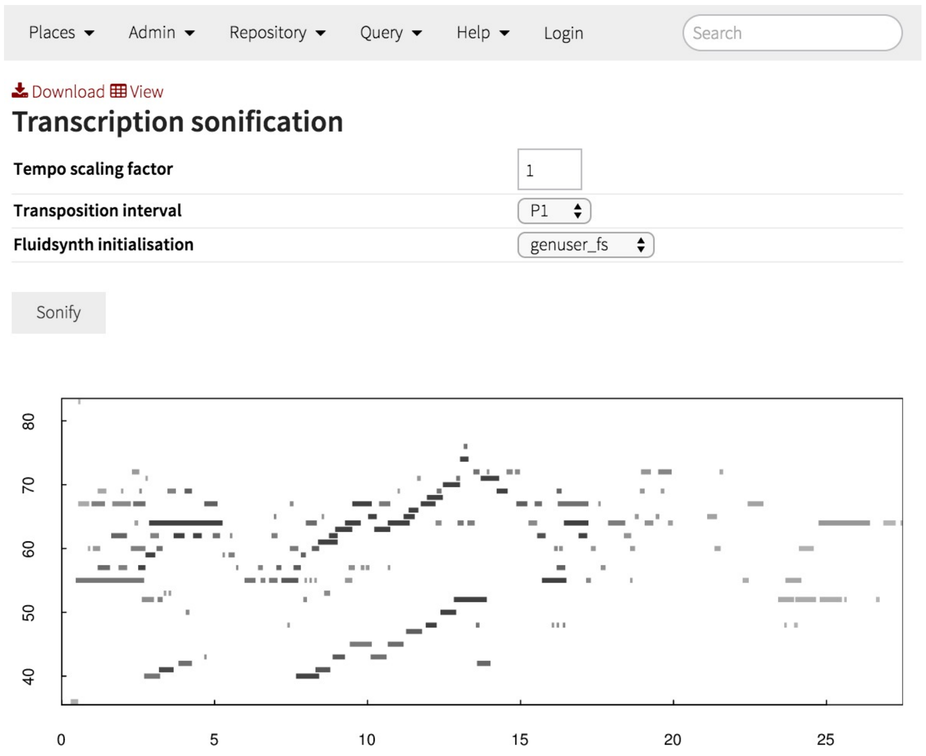
3. Real libraries and applications

3.1. Digital Music Laboratory

The Digital Music Laboratory (DML[1]) is a system for doing large scale collaborative computational musicology, bringing together a collection of audio recordings and symbolic scores, their metadata, a collection of computational analysis methods, and a database of computation results. It consists of an information and computation management server providing web APIs, a browser, a Javascript web application for exploration and visualisation, and a web-based programming environment, a SWISH instance [22], for more flexible, free form experimentation. The system uses *Real* in several ways:

- The browser, based on the SWI Prolog Semantic Web application framework Cliopatria [26] uses *R* to produce high quality scalable vector graphics, such as a pitch histogram for a musical scoreor a piano-roll representation of an automatic transcription[2], to be included in web pages describing musical resources (see Fig. 1).

[1] `http://dml.city.ac.uk`
[2] `http://mirg.city.ac.uk/cp/browse/list_resource?r=file://vamp/_Audio_Analysis/Chopin/silvet_settings_fast_nonfinetune_allinstruments.n3_9d782/026A-1CL0016611XX-0100A0_vamp_silvet_silvet_notes.csv`



Transcription sonification

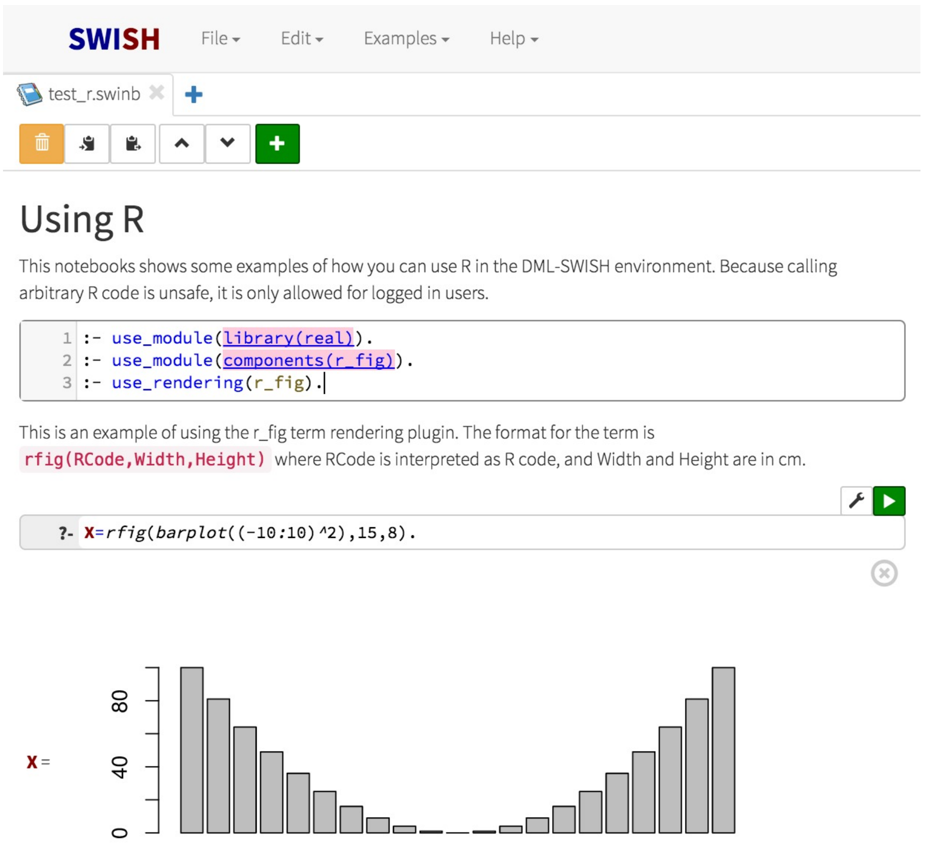
Tempo scaling factor	1
Transposition interval	P1
Fluidsynth initialisation	genuser_fs

Figure 1: An example of *R* graphics embedded in a web page on the Digital Music Laboratory website. This page is associated with the results of a music transcription algorithm, whose result is visualised as a piano-roll using *R*'s `rect()` plotting function.

- The computation system uses *R* for doing statistical analysis on the contents of the database, such as music metadata or results from previous computations.
- The programming environment allows users to compose their own computations, possibly using *Real* for numerical analysis or generating plots.

An example from DML is shown in Fig. 2 which displays part of a browser window running SWISH, with a simple *R* computation and rendering.[3] Because of the web-based, collaborative nature of the system, it is important that it be able to handle multiple concurrent requests for *R* computations. Thus, it uses *Real*'s ability to start a singe *R* server thread, which then receives and serialises

[3]`http://mirg.city.ac.uk/cp/swish/p/test_r.swinb`, but please note that *R* computations and graphics in the SWISH environment are restricted to logged-in users, for security reasons.



Using R

This notebooks shows some examples of how you can use R in the DML-SWISH environment. Because calling arbitrary R code is unsafe, it is only allowed for logged in users.

```
:- use_module(library(real)).
:- use_module(components(r_fig)).
:- use_rendering(r_fig).
```

This is an example of using the r_fig term rendering plugin. The format for the term is `rfig(RCode,Width,Height)` where RCode is interpreted as R code, and Width and Height are in cm.

```
?- X=rfig(barplot((-10:10)^2),15,8).
```

Figure 2: An example of how R can be used inside the SWISH web-based Prolog development environment. A SWISH 'term rendering' plugin is included to allow certain terms to be recognised as plot specifications and rendered accordingly.

all R computations.

In comparison with previous versions of the system which used Matlab's engine API to communicate with a separate *Matlab* process, via the *plml* package [1], the lower overhead of communicating with *Real*'s in-process embedded R yields much better performance when numerous relatively small computations are required.

Graphical output from R is included in web pages or the SWISH programming environment ultimately as an HTML image element referring to a URL on the server that will dynamically generate the plot and return it as an SVG stream. The handler for this URL takes literal Prolog code as a parameter, checks that this code is safe to run using the SWI Prolog *sandbox* library, runs the code to create the plot, saves the graphic in a temporary directory, and replies with the contents of the file. A simplified view of the code, which can handle several output formats, is shown in Fig. 3.

```
% this is the HTTP request handler
r_figure_render(Request)  :-
  % ...code to extract parameters Code, Fmt, W, and H omitted...
  sandbox:safe_goal(Code),
  with_temp_dir(Dir, (
    file_name_extension(tmpfig,  Fmt, Filename),
    directory_file_path(Dir,    Filename, Path),
    with_mutex(r_plot,  print_fig(Fmt,  Code, Path, [size(W,H)])),
    reply_file(Path,   Fmt))). % sends HTTP reply with correct MIME type

% run Code and save graphic in Path in requested format
print_fig(Fmt,  Code, Path, Opts) :-
  option(size(Width,Height),  Opts),
  dev(Fmt, Path, Width, Height, Dev),
  setup_call_cleanup(r(Dev),  once(Code), r('dev.off()')).

% table of R codes to create a graphics context in a given format.
dev(pdf,  Path,  W, H, pdf(+Path, width= W, height= H)).
dev(eps,  Path,  W, H, cairo_ps(+Path, width= W, height= H)).
dev(svg,  Path,  W, H, svg(+Path, width= W, height= H)).
```

Figure 3: Simplified extract of Prolog code used to serve graphics generated in *R* in response to HTTP requests.

3.2. Computational Biology

The development of *Real* was strongly motivated by the pervasiveness of *R* code in bioinformatics, with many of the numerous relevant *R* libraries being part of Bioconductor [7]. *Real* has been used in a number of projects including a complete pipeline for the functional analysis of SILAC, proteomic datasets [16, 27]. Within the context of this project a number of useful components have been developed as independent Prolog libraries that allow users to tap on *R* graphics and *R* functions without writing any *R* code. In what follows we describe three of these libraries. The emphasis here is to assist Prolog programmers that are not familiar with *R* to take advantage without having to learn *R*. If later on they become more familiar with the language they can take fuller advantage of *R* via *Real*. The penetration of Prolog into bioinformatics have been also been advocated by more holistic approaches [13]. However the approach we have taken here is to take advantage of SWI's package manager to produce a number of independent libraries that can be used in a number of application areas.

3.2.1. Basic Real *library*

An introductory library to using *R* is *b_real*. It is based on basic *R* calls via *Real* and it contains a collection of predicates that aim to provide a Prolog based interface to a number of simple *R* tasks. The target audience is Prolog users that have little or no previous experience with *R*. Providing interface predicates

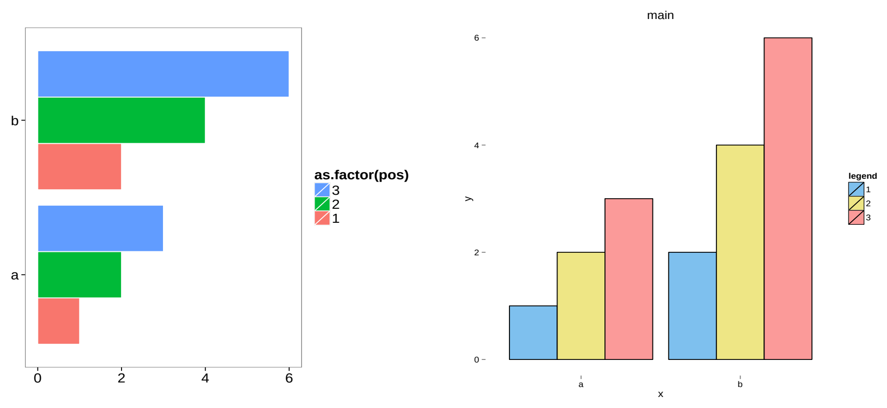


Figure 4: ggplot2 based bar plots. Left: with default options. Right: a number of options have altered elements of the plot.

to *R* functions allows: use of untyped arguments or Prolog friendly arguments, ability to use the more familiar Prolog syntax and better control over side-effects of the calls. The predicates described here can use the basic functionality of the underlying *R* functions and can adjust some of the behaviour entirely in Prolog, while allowing arbitrary option passing to users with some familiarity with *R*.

Bar plots are basic plots that can present comparative information in a intuitive manner. Here we present a Prolog interface to *ggplot2* [20]. In its most general form, the predicate *gg_bar_plot*/2 displays a number of grouped measurements such as, for instance, the cpu-times of a number of machine learning algorithms ran on a number of datasets. The following query, produces the plot in the LHS of Fig. 4.

```
?- Pairs = [a-[1,2,3], b-[2,4,6]],   gg_bar_plot(Pairs,[]).
```

ggplot2 is a complex piece of software able to display many types of plots while *gg_bar_plot*/2 only accessing the bar plotting part. Within this, a number of plot elements can be controlled with Prolog options passed in the second argument. The following query changes elements such as the colour of the drawing pen (*black*) the labels (*x*,*y* and *main*), legend title and fill colours, producing the plot in the RHS of Fig. 4.

```
?- Pairs = [a-[1,2,3], b-[2,4,6]],
   Opts  = [ geom_bar_draw_colour(black),
             fill_colours(["skyblue2","khaki2","#FB9A99"]),
             flip(false),    labels(x,y,main),
             legend_title(legend)    ],
   gg_bar_plot(   Pairs,  Opts ).
```

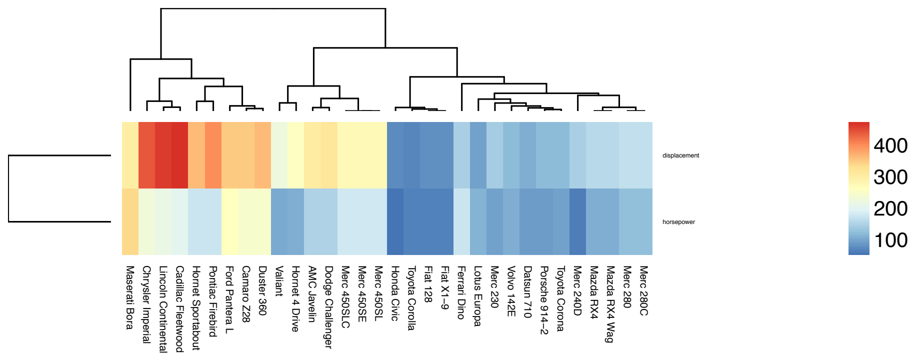


Figure 5: Heatmap generation with aheatmap() from package NMF.

Heatmap drawing functions are ubiquitous in *R*. *b_real* provides an interface to the *aheatmap* function. In addition to some simple option mapping *aheatmap*/2 provides polymorphic support for the first argument which could be a matrix *R* variable or a Prolog representation of one. The following code uses the *mtcars* example dataset, from which it plots a heatmap of two variables: *hp* (horsepower) and *disp* (displacement).

$?-\ MtC \leftarrow as.list(mtcars),\ memberchk(hp = HP, MtC),$
$\quad memberchk(disp = Disp, MtC),\ x \leftarrow [HP, Disp],$
$\quad rownames(x) \leftarrow c(\text{"horsepower","displacement"}),$
$\quad \leftarrow aheatmap(x).$

3.2.2. Weighted graphs

R has a number of plotting functions for drawing graphs formed of nodes and edges. Two of these are `igraph()` and `qgraph()`. The latter being based on the former with some extra options and facilities for grouping nodes. The Prolog pack *wgraph* provides a uniform Prolog interface to these two *R* libraries. A plot with the default renderings can be easily drawn from a list representing the graph connections and the weights on the edges:

$?-\ G\ = [1{-}2{:}200,\ 2{-}3{:}400,\ 2{-}4{:}300],$
$\quad wgraph\_plot(G, [\,]).$

A set of Prolog options that control the choice of the drawing function and basic parameters of the graph, and which work irrespective of the drawing function can be provided in the second argument of *wgraph_plot*/2. In the following example `igraph()` is passed the size of nodes to use, the degree at which the node labels should be displayed and the distance of the label from the node edge. The resulting graph is shown in the RHS of Fig. 6.

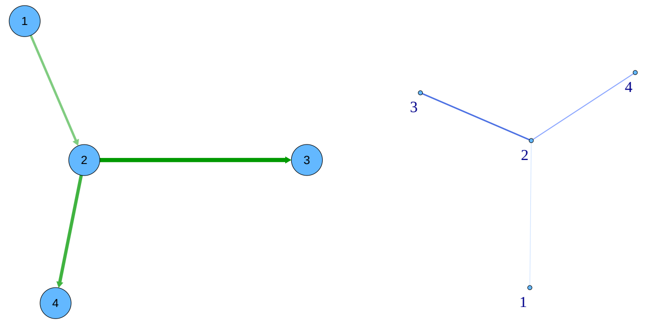


Figure 6: Graphs generated by *wgraph_plot*/2. Left: plot uses default rendering with *qgraph()* call. Right: render changed to *igraph()* and a number of options specialised the output.

```
?- G =[1-2:200, 2-3:400, 2-4:300],
   Opts =[ plotter(igraph),
           label_distance(-1),
           label_degree(2),
        node_size(4) ],
   wgraph_plot( G, Opts ).
```

Weighed graphs are a popular means of displaying biological information such as protein-protein interactions. Databases such as *String* [17] hold interaction relations over proteins that are either experimentally verified or inferred algorithmically. Depending on the strength of the evidence and corroborating sources, String assigns a degree of belief $(0, 999)$ on each possible edge. Given a set of proteins or genes it is thus possible to visualise their inter-connectivity in *String* as weighed graphs with the width of the edges proportional to the degree of belief on the edge's existence. The *Gene Ontology* (*GO*, [18]) annotates genes with a number of terms that belong in one of the ontology's three categories: cellular component, molecular function and biological process. By combining *GO* and *String* we can visualise the physical interactions between genes in a *GO* term as a weighed graph.

Prolog predicates for both *GO* and *String* are provided by *bio_db* [3]. This library implements predicates that map database tables from a number of curated databases to a single library relation. For demonstrating *wgraph's* capabilities two predicates are of interest: *map_gont_gont_symb*(*GOterm*,*Symbol*) is the set membership relation of *Symbols* to gene ontology terms (*GOterm*). *edge_string_hs_symb*(*Symbol1*,*Symbol2*, *Weight*) holds the *String* defined gene interactions with an associated belief measure (*Weight*). Having access to these predicates and with Prolog's powerful search execution model graphs that show the interactions of genes within a GO term can be constructed and easily visualised with *wgraph*. An example of such a network is shown in 7 . The code for

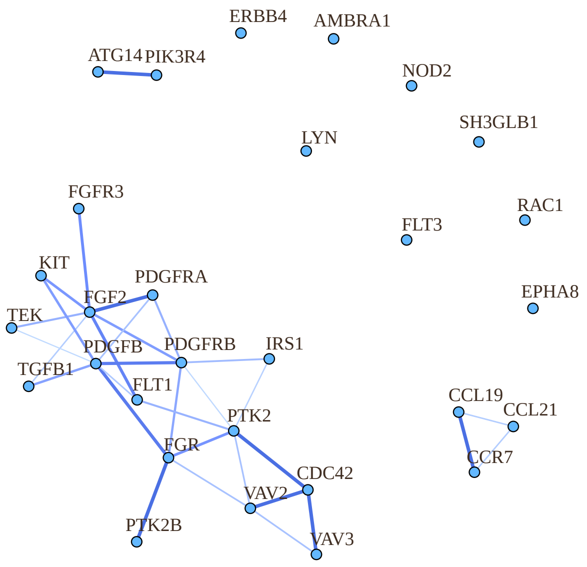


Figure 7: Weighed graph for gene ontology term, *GO*:0043552, *positive regulation of phosphatidylinositol 3-kinase activity.*

reproducing this is succinct and typical of the high level programming that can be achieved with Prolog.

```
?- use_module(library(bio_db)),  use_module(library(wgraph)).
?- findall(   Symb, map_gont_gont_symb('GO:0043552',Symb), Symbs ),
    findall(   Symb1-Symb2:Weight, (
                                   member(Symb1,Symbs),
                                   member(Symb2,Symbs),
                                   edge_string_hs_symb(Symb1,Symb2,Weight)
                                      ),
                                        Graph ),
   wgraph_plot( Graph ).
```

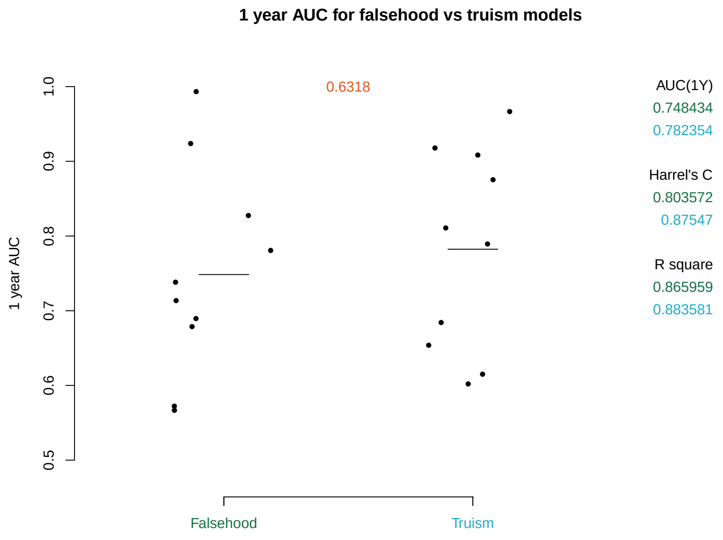


Figure 8: Comparative performance of two simple learners. The first comparative metric is used to plot the dots where any additional metrics are averaged over the K folds and displayed on the right edge.

3.2.3. Machine learning utilities

Machine learning utilities, *mlu*, is a collection of machine learning predicates that allow users to run machine learning experiments and predictors on the learned models. The library does not implement the learners themselves but deals with tasks such K-fold cross validation and plotting of predicted values that allow evaluation of performance for a number of algorithms. The learners themselves can be either other Prolog predicates such as Aleph (*pack(aleph)*, [15] and Bims (*pack(bims)*, [2]) or one of the plethora of R machine learning and statistical inference functions. The following code declares two deterministic learners, *truism*/2 and *falsehood*/2, which always learn *t$$rue* and *f$$alse* respectively. We then define predictors that generate 4 measures of accuracy for each model. In this example the predictors are based on random values that are biased by the type of input model with **true** models fairing slightly better.

Using *k_fold_comparative_statistic*/6 a number of learners can be compared by running each learner on the same K fold exhaustive partition of the data and obtaining performance statistics. The predictions can then be tabulated or plotted for visual comparison of performance. Figure 8 shows a comparative plot of the performance for the two simple learners presented below. In the plot, the first predictive value is used to draw juxtaposed dots on a scatter plot with all additional predictive measures averaged over the 10 folds and displayed as mean values on the right ledger of the plot.

```
truism(_,true).
falsehood(_,false).

bias(false,Data,c(Rnd,Rnd,Rnd2,Rnd3)) :-
  random(0.5,1,Rnd),
  random(0.65,1,Rnd2),
  random(0.7,1,Rnd3).
bias(true,Data,c(Rnd,Rnd,Rnd2,Rnd3)) :-
  random(0.6,1,Rnd),
  random(0.75,1,Rnd2),
  random(0.8,1,Rnd3).

?- numlist(1,10,Data),
   Learners =[falsehood,truism],
   Predictor= bias,
   Opts =[post(jitter),
     statistic_names(['1 year AUC','AUC(1Y)','Harrel\'s C','R square'])
     ],
  k_fold_comparative_statistic(Data,Learners,Predictor,_Ms,_Sts,Opts).
```

Although the example given here is a simple one, similar analysis can be applied to a variety of learning contexts such as survival analysis (for example with the *coxph*() function) where metrics such as AUC for 1 year survival, Harrrel's concordance index and R^2 can be used to document the relevant performance of alternative learners. We also plan to include bootstrapping in *mlu* along with more comparative plots.

3.3. Availability

The four libraries discussed here: *Real*, *b_real*, *wgraph* and *mlu* are available as SWI-Prolog packages[4] which can be installed easily from within SWI-Prolog. To download and install *Real* the user needs to query with:

```
?- install_pack(real).
```

More information including documentation and full sources are available from the project's home page [5]. These libraries have been developed in the context of Prolog based bioinformatics [3, 5] but are made available as stand alone libraries as they can be of use in a variety of other application areas. The package manager enables easy installation and a central point at which user-contributed libraries can be found.

[4] `http://swi-prolog.org/pack/list`
[5] `http://stoics.org.uk/~nicos/sware/real/`

4. Conclusions and Future Work

We presented a number of recent advances in *Real* and we have shown how developments in Prolog syntax allow *R* calling code to blend naturally into Prolog code. The resulting syntax provides a powerful platform for accessing the extensive collection of open source *R* code. As a result, *Real* can have a strong positive influence onto the penetration of Prolog to new application areas such as bioinformatics and the semantic web. With version 2, *Real* has reached a new level of maturity including facilities for using *R* in web-servers. In addition, we highlighted predicates from sister packages. As with *Real* itself, these are freely available and can be easily installed via the SWI-Prolog package manager. In the future we plan to work towards suggesting internal ways for Prolog to work better, or more confluent to *R*, with *NA* values and infinity.

With regard to web applications, the *R* server thread serialises *R* computations at the level of individual calls to the embedded *R* API. This is fine for 'pure' functional operations that have no side effects, that is, do not change the state of the *R* session nor do any input or output. However, it is still possible for operations that require several calls with side-effects to impede on each other's operation. This is the case for graphical plots that involve several steps, each of which may add elements to a plot, before the final step of writing the resulting image to a file. If two threads attempted to create two such plots concurrently, the single graphics window would become corrupted. The DML server described above avoids this by using an SWI Prolog mutex to protect the *R* graphics HTTP request handler, essentially turning the entire plotting sequence into an atomic operation. Another, possibly more elegant solution not requiring a mutex, would be to allow the *R* server thread to receive requests to execute Prolog goals, rather than just atomic *R* computations. This would allow each Prolog goal, including multiple calls to *R*, to complete atomically, safe in the knowledge that no other *R* computations will intervene. Current work by the developers of SWISH [22], the on-line portal for SWI-Prolog, concentrates on : (a) isolating interactions for each distinct user, (b) capturing the input/output of the underlying *R* sessions, and (c) confining *R* to a safe subset, such as say to disallowing operating system interactions.

Real has been used in a number of projects in the areas of web programming and bioinformatics, and has a steady stream of downloads via SWI-Prolog's package manager. With the enhanced level of integration, *Real* is becoming a powerful hybrid programming language, which combines the clear computational model of Prolog with the powerful statistical code base of *R*.

Acknowledgements

We would like to thank Jan Wielemaker, main developer of SWI-Prolog, for technical advice, and for providing a robust, open source Prolog, as well as for comments on earlier drafts of this paper. We also thank Vítor Santos Costa for earlier and continuing support to the development of *Real* and the anonymous reviewers for their insightful observations.

References

[1] Samer Abdallah. plml: Prolog-Matlab bridge, 2015. `http://www.swi-prolog.org/pack/list?p=plml`.

[2] Nicos Angelopoulos and James Cussens. Bayesian learning of Bayesian networks with informative priors. *Journal of Annals of Mathematics and Artificial Intelligence*, 54(1-3):53–98, 2008.

[3] Nicos Angelopoulos and Georgios Giamas. A logical approach to working with biological databases. In *International Conference of Logic Programming*, Cork, Ireland, September 2015. Technical communication.

[4] Nicos Angelopoulos, Vítor Santos Costa, João Azevedo, Jan Wielemaker, Rui Camacho, and Lodewyk Wessels. Integrative functional statistics in logic programming. In *Proc. of Practical Aspects of Declarative Languages*, volume 7752 of *LNCS*, pages 190–205, Rome, Italy, Jan. 2013.

[5] Sander Canisius, Nicos Angelopoulos, and Lodewyk Wessels. ProSQLite: Prolog file based databases via an SQLite interface. In *Proc. of Practical Aspects of Declarative Languages*, volume 7752 of *LNCS*, pages 222–7, 2013.

[6] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12:5–34, 1 2012.

[7] Robert C. Gentleman, Vincent J. Carey, Douglas M. Bates, and others. Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5:R80, 2004.

[8] Robert Tibshirani Jerome Friedman, Trevor Hastie. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33:1–22, 2010.

[9] Alexandros Karatzoglou, Alex Smola, Kurt Hornik, and Achim Zeileis. kernlab – an S4 package for kernel methods in R. *Journal of Statistical Software*, 11(9): 1–20, 2004.

[10] Torbjörn Lager and Jan Wielemaker. Pengines: Web logic programming made easy. In *International Conference of Logic Programming*, 2014.

[11] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002.

[12] David MacIntyre, Manju Chandiramani, Yun S Lee, Lindsay Kindinger, Ann Smith, Nicos Angelopoulos, Benjamin C. Lehne, Shankari Arulkumaran, Richard Brown, Tiong Ghee Teoh, Elaine Holmes, Jeremy K. Nicholson, Julian Marchesi, and Phillip R. Bennett. The vaginal microbiome during pregnancy and the post-partum period in a european population. *Scientific Reports*, 5:Article number: 8988, 2015.

[13] Chris Mungall. Experiences using logic programming in bioinformatics. In *Logic Programming*, pages 1–21. Springer Berlin Heidelberg, 2009.

[14] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, 2016. `http://www.R-project.org/`.

[15] Ashwin Srinivasan. *The Aleph Manual.* University of Oxford, 2004.

[16] Justin Stebbing, Hua Zhang, Yichen Xu, Grothey Arnhild, Ajuh Paul, Nicos Angelopoulos, and Georgios Giamas. Reprogramming of the tyrosine kinase-regulated proteome in breast cancer by combined use of RNAi and SILAC quantitative proteomics. *Molecular & Cellular Proteomics*, 14(9):2479–92, 2015.

[17] Damian Szklarczyk, Andrea Franceschini, Stefan Wyder, Kristoffer Forslund, Davide Heller, Jaime Huerta-Cepas, Milan Simonovic, Alexander Roth, Alberto Santos, Kalliopi P. Tsafou, Michael Kuhn, Peer Bork, Lars J. Jensen, and Christian von Mering. String v10: proteinprotein interaction networks, integrated over the tree of life. *Nucleic Acids Research*, 43(D1):D447–D452, 2015.

[18] The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25(1):25–9, May 2000.

[19] Terry Therneau. *A Package for Survival Analysis in S*, 2015. version 2.38.

[20] Hadley Wickham. *ggplot2: elegant graphics for data analysis.* Springer New York, 2009. ISBN 978-0-387-98140-6. `http://had.co.nz/ggplot2`.

[21] Jan Wielemaker. SWI-Prolog ODBC interface, 2014. `http://www.swi-prolog.org/pldoc/package/odbc.html`.

[22] Jan Wielemaker. SWISH: SWI-Prolog for SHaring, 2015. `http://swish.swi-prolog.org/`.

[23] Jan Wielemaker and Vítor Santos Costa. On the portability of Prolog applications. In *Practical aspects of Declarative Languages*, pages 69–83, 2011.

[24] Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. SWI-Prolog and the Web. *Theory and Practice of Logic Prog.*, 8(3):363–92, 2008.

[25] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[26] Jan Wielemaker, Wouter Beek, Michiel Hildebrand, and Jacco van Ossenbruggen. Cliopatria: A SWI Prolog infrastructure for the Semantic Web. *Semantic Web Journal*, pages 1–13, 2015.

[27] Hua Zhang, Nicos Angelopoulos, Yichen Xu, Arnhild Grothey, Joao Nunes, Justin Stebbing, and Georgios Giamas. Proteomic profile of KSR1-regulated signaling in response to genotoxic agents in breast cancer. *Breast Cancer Research and Treatment*, 151(3):555–568, June 2015.