Foreign Library Interface

by Daniel Adler

December 13, 2011

Abstract

We present an improved Foreign Function Interface (FFI) for R to call arbitary native functions without the need for C wrapper code. Further we discuss a dynamic linkage framework for binding standard C libraries to R across platforms using a universal type information format. The package **rdyncall** comprises the framework and an initial repository of cross-platform bindings for standard libraries such as (legacy and modern) *OpenGL*, the family of *SDL* libraries and *Expat*. The package enables system-level programming using the R language; sample applications are given in the article. We outline the underlying automation tool-chain that extracts cross-platform bindings from C headers, making the repository extendable and open for library developers.

1 Introduction

We present an improved Foreign Function Interface (FFI) for R that significantly reduces the amount of C wrapper code needed to interface with C. We also introduce a *dynamic* linkage that binds the C interface of a pre-compiled library (*as a whole*) to an interpreted programming environment (Ousterhout, 1997) such as R - hence th name *Foreign Library Interface*. Table 1 gives a list of the C libraries currently supported across major R platforms. For each library supported, abstract interface specifications are declared in a compact platform-neutral text-based format stored in so-called *DynPort* files on a local repository.

R (Ross Ihaka and Robert Gentleman, 1996) was choosen as the first language to implement a proof-of-concept implementation for this approach. This article describes the **rdyncall** package which implements a complete toolkit of low-level facilities that can be used as an alternative FFI to interface with the C programming language. And further, it enables direct and quick access to the common C libraries from R without compilation.

The project was motivated by the fact that high-quality software solutions implemented in portable C are often not available in interpreter-based languages such as R. The pool of freely available C libraries is quite large and represents an invaluable resource for software development. For example, OpenGL (Board et al., 2005) is the most portable and standard interface to accelerated graphics hardware for developing real-time graphics software. The combination of OpenGL with the *Simple DirectMedia Layer* (SDL) (Lantinga and et al, 2011) core and extension libraries offers a foundation framework for developing interactive multimedia applications that can run on a multitude of platforms. Other libraries such as the Expat XML Parser (Clark, 2011) provide a parser framework for processing very large XML documents. And even the C library of R contains high-quality statistical functions that are useful in context of other languages as well.

To make use of these libraries within high-level languages, *language bindings* to the library must be written as an extension to the language, a task that requires deep familiarity of the internals of both the library and the interpreter. Depending on the complexity of the library,

lib/dynport	description	functions	constants	aggregate types
`gl`	opengl	337	3253	-
`glu`	opengl utility	59	154	-
`r`	r library	238	700	27
`sdl`	audio/video/ui abstraction	203	465	51
`sdl_image`	pixel format loaders	29	-	-
`sdl_mixer`	music format loaders and playing	63	12	-
`sdl_ttf`	font format loaders	35	9	-
`cuda`	gpu programming	387	665	84
`expat`	xml parsing framework	65	70	-
`glew`	gl extensions	1465	-	-
`gl3`	opengl 3 (strict)	324	838	1
`opencl`	gpu programming	78	260	10
`stdio`	standard i/o	76	3	-

Table 1: overview of available dynports for portable c libraries

the amount of work needed to wrap the interface can be very large (Table 1 gives the counts of functions, constants and types that need to be wrapped). Rather than having to write a separate binding for each *library and language* combination, we research a dynamic binding approach that is adaptable to interpreters and works cross-platform without additional compilation of wrapper layers. Once the binding specification for a library has been specified, that library becomes automatically accessible to all interpreters that implement such a framework outlined here. Extension techniques offered by the language interpreter, such as a *Foreign Function Interface* (FFI), are the fundamental technology for bridging the dynamic interpreter with statically pre-compiled code.

In the case of R the built-in FFI function `.C` provides a fairly basic call gate to C code with strong limitations; additional wrapper code has to be written in addition to interface with standard C libraries. **rdyncall** contributes an improved FFI for R that offers a *flexible* and *type-safe* interface with support for almost all C types without requiring additional C wrappers.

Based on this FFI, the package contains a proof-of-concept implementation of a *Foreign Library Interface* that enables *direct* and *dynamic* interoperability with foreign C Libraries (including shared library code and the Application Programming Interface specified in C headers) from within the R interpreter. For each C library supported, abstract interface specification are declared in a compact platform-neutral text-based format stored in a so-called *DynPort* file located in a local repository within the package. Table 1 gives a sample list of available bindings that come with the package.

Users gain access to C libraries from R using the front-end function `dynport(`*portname*`)`, which processes a *DynPort* file to load the C library[1], and wrap the C interface as a newly attached R name space that uses the same symbolic names of the C API. R code that uses C interfaces via *DynPort*s might look very familiar to C user code.

This article motivates the topic with a comparison of the built-in and contributed FFI by means of a simple use case. This leads to a detailed description of the improved FFI. Then follows an overview of the package and a brief tour through the framework with details on the handling of foreign C data types and wrapping R functions as callbacks. Two sample applications are given using OpenGL, SDL and Expat. The article ends with a brief description

[1]Pre-compiled libraries need to be installed, OS-specific installation notes are given in the documentation of the package.

of the implementation based on C libraries from the *DynCall* project (Adler and Philipp, 2011) and the tool-chain that was used to create the repository of *DynPort* files.

2 Foreign Function Interfaces

FFIs provide the backbone of a language to interface with foreign code. Depending on the design of this service, it can largely unburden developers from writing additional wrapper code. In this section, we compare the built-in FFI with the improved FFI provided by **rdyncall** using a simple example that sketches the different work flow paths for making an R binding to a function from a foreign C library.

2.1 FFI of base R

Suppose that we wish to invoke the C function `sqrt` of the C Standard Math library. The function is declared as follows in C:

```
double sqrt(double x);
```

R offers a number of functions to call pre-compiled code from within the R interpreter. While `.Call` and `.External` are designed for interoperability with *extension* code, `.C` and `.Fortran` seem to offer the most low-level interoperability with *foreign* code. But `.C` has also very strict conversion rules and strong limitations regarding argument and return-types: `.C` passes R arguments as C pointers and C return types are not supported, so only C `void` functions, which are procedures, can be called. Given these limitations, we are not able to invoke the foreign `sqrt` function directly and need some intermediate wrapper code written in C that obeys the rules of the `.C` interface:

```
#include <math.h>
void R_C_sqrt(double * ptr_to_x)
{
  double x = ptr_to_x[0], ans;
  ans = sqrt(x);
  ptr_to_x[0] = ans;
}
```

We assume that the wrapper code is deployed as a shared library in a package named *testsqrt* which links to the C math library. [2]. Then we load the *testsqrt* package and call the C wrapper function directly via `.C`.

```
> library(testsqrt)
> .C("R_C_sqrt", 144, PACKAGE="testsqrt")
[[1]]
[1] 12
```

To make `sqrt` available as a public function, an additional R wrapper layer is added, that does type-safety checks before issuing the `.C` call.

```
sqrtViaC <- function(x)
{
  x <- as.numeric(x) # type(x) should be C double.
  # make sure length > 0:
```

[2] We omit here the details such as registering C functions which is described in detail in the R Manual '*Writing R Extensions*' (R Development Core Team, 2010).

```
  length(x) <- max(1, length(x))
  .C("R_C_sqrt", x, PACKAGE="example")
}
```

As an alternative, R also provides high-level C extension interfaces such as `.Call` and `.External`, that give access to R internals at C level and enable to make type-safety checks within C:

```
#include <R.h>
#include <Rinternals.h>
#include <math.h>
SEXP R_Call_sqrt(SEXP x)
{
  SEXP ans = R_NilValue, tmp;
  PROTECT( tmp = coerceVector(x, REALSXP) );
  if (LENGTH(tmp) > 0) {
    double y = REAL(tmp)[0], result;
    result = sqrt(y);
    ans = ScalarReal(result);
  }
  UNPROTECT(1);
  return ans;
}
```

Now the corresponding R wrapper shrinks into a simple delegate:

```
> sqrtViaCall <- function(x)
+ .Call("R_Call_sqrt", x, PACKAGE="example")
```

The third alternative, via `.External`, is omitted here; it has a different argument passing scheme, but the C and R wrapper implementations would look very similar.

We can conclude that - in realistic settings - the built-in FFI of R almost always needs support by a wrapper layer written in C. The "foreign" in FFI is in fact relegated to the C wrapper layer.

Moreover the R FFI can be viewed as an *extension* interface for calling pre-compiled code written in a *foreign* language within the context of the R implementation, rather than a direct invocation interface for code from a *foreign* context such as an ordinary C library.

2.2 FFI of rdyncall

rdyncall provides an improved FFI for R that is accessible via the function `.dyncall`. In contrast to the built-in R FFI which uses a C wrapper layer, the `sqrt` function is invoked dynamically and directly by the interpreter at run-time. Whereas the C math library was loaded implicitly via the example package, it now has to be loaded explicitly.

R offers functions to deal with shared libraries at run-time, but the location has to be specified as an absolute pathname which is platform-specific. For now, let us assume that the example is done on Mac OS X where the C math library is located at '/usr/lib/libm.dylib'. A platform-portable solution is discussed in the next section on *Portable loading of shared library*.

```
> libm <- dyn.load("/usr/lib/libm.dylib")
> sqrtAddr <- libm$sqrt$address
```

We first need to load the R package **rdyncall**:

Type	Sign.	Type	Sign.
`void`	`v`	`bool`	`B`
`char`	`c`	`unsigned char`	`C`
`short`	`s`	`unsigned short`	`S`
`int`	`i`	`unsigned int`	`I`
`long`	`j`	`unsigned long`	`J`
`long long`	`l`	`unsigned long long`	`L`
`float`	`f`	`double`	`d`
`void*`	`p`	`struct` *name* `*`	`*<`*name*`>`
type`*`	`*...`	`const char*`	`Z`

Table 2: C/C++ Types and Signatures

```
> library(rdyncall)
```

Finally, we invoke the foreign C function `sqrt` *directly* via `.dyncall`:

```
> .dyncall(sqrtAddr, "d)d", 144)
[1] 12
```

Let us review the last call, as it pinpoints the core solution for a direct invocation of foreign code within R: The first argument specifies the address of the foreign code, given as an external pointer. The second argument is a *call signature* that specifies the argument- and return types of the target C function. This string `"d)d"` specifies that the foreign function expects a `double` scalar argument and returns a `double` scalar value in correspondence to the C declaration of `sqrt`. Arguments following the call signature are passed to the foreign function using the call signature for type-safe conversion to C types. In this case we pass `144` as a C `double` argument type as first argument and receive a C `double` value converted to an R `numeric`.

2.3 Call Signatures

The introduction of a type descriptor for foreign functions is a key component that makes the FFI flexible and type-safe. The format of the call signature has the following pattern:

argument-types `')'` *return-type*

The signature can be derived from the C function declaration: Argument types are specified first, in a left-to-right order, and are terminated by the `')'` symbol followed by a single return type signature.

Almost all fundamental C types are supported and there is no real restriction regarding the number of arguments supported to issue a call. Table 2 gives an overview of supported C types and the corresponding text encoding; Table 3 provides some examples of C functions and call signatures.

Now, let us define a public and type-safe R wrapper function that hides the details of the foreign function call by passing the formal argument place holder "`...`" as third argument to `.dyncall`:

```
> sqrtViaDynCall <- function(...)
+ .dyncall(sqrtAddress, "d)d", ...)
```

C function declaration	dyncall type signature
`void rsort_with_index(double*,int*,int n)`	`*d*ii)v`
`SDL_Surface * SDL_SetVideoMode(int,int,int,Uint32_t)`	`iiiI)*<SDL_Surface>`
`void glClear(GLfloat,GLfloat,GLfloat,GLfloat)`	`ffff)v`

Table 3: Some examples of C functions and corresponding signatures

Although there is no further guard code, this interface is type-safe and the user can do no harm by inadvertently using a wrong set and/or type of arguments due to the built-in type-checks. Compared to the R wrapper code using `.C`, no explicit cast of the arguments via `as.numeric` is required, because automatic coercion rules for fundamental types are implemented as dictated by the call signature. For example, `integer` R values are implicitly casted to `double` automatically:

```
> sqrtViaDyncall(144L)
[1] 12
```

A certain level of type-safety is achieved here as well: All arguments to be passed to C are first checked against the call signature. If any incompatibility is detected, such as a wrong number of arguments, empty atomic vectors or incompatible type mappings, the invocation is aborted and an error is reported before risking an application crash:

```
> sqrtViaDyncall(1,2)
Error in .dyncall(sqrtAddress, "d)d", ...) :
  Too many arguments for signature 'd)d'.
> sqrtViaDyncall()
Error in .dyncall(sqrtAddress, "d)d", ...) :
  Not enough arguments
    for function-call signature 'd)d'.
> sqrtViaDyncall(NULL)
Error in .dyncall(sqrtAddress, "d)d", ...) :
  Argument type mismatch at position 1:
    expected double convertible value
> sqrtViaDyncall("144")
Error in .dyncall(sqrtAddress, "d)d", ...) :
  Argument type mismatch at position 1:
    expected double convertible value
```

In contrast to the R FFI, where the argument conversion is dictated solely by the R argument type at call-time in a one-way fashion, the introduction of an additional specification with a call signature gives several advantages.

- Almost all possible C functions can be invoked by a single interface; no additional C wrapper is required.
- The built-in type-safety checks of passed arguments enhance stability and reduce assertion code in R wrappers significantly.
- A single call signature can work across platforms, given that the C function type remains constant across platforms.
- Given that our FFI is implemented in multiple languages, call signatures represent a portable type description for C libraries.

3 Package Overview

Besides dynamic calling of foreign code, the package provides essential facilities for interoperability between the R and C programming languages. A high-level overview of components that make up the package is given in Figure 1.

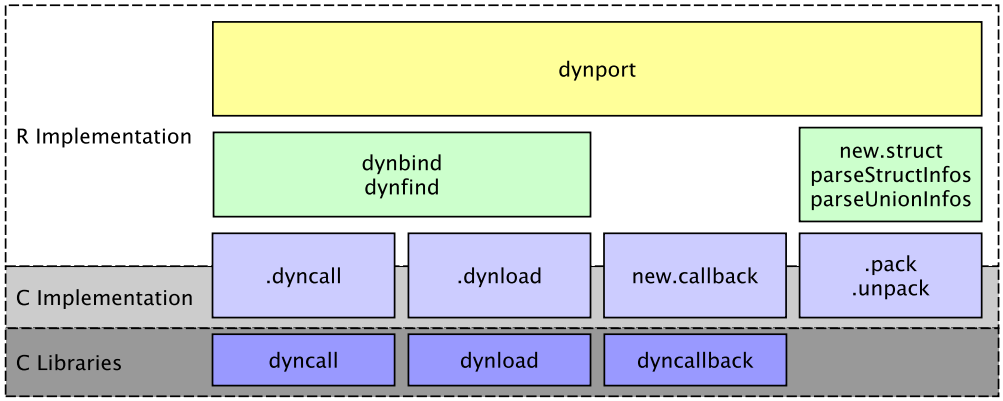

Figure 1: Package Overview

We already described the `.dyncall` FFI. It follows a brief description of portable loading of shared libraries using `dynfind`, installation of wrappers via `dynbind`, handling of foreign data types via `new.struct` and wrapping of R functions as C callbacks via `new.callback`. Finally the high-level `dynport` interface for accessing *whole* C libraries is briefly discussed. The technical details at low-level of some components are described briefly in the section *Architecture*.

3.1 Portable loading of shared libraries

The *portable* loading of shared libraries across platforms is not trivial because the file path is different in Operating-Systems (OS). Referring back to the previous example, to load a particular library in a portable fashion, one would have to check the platform to locate the C library.[3]

Although there is variation among the OSs, library file paths and search patterns have common structures. For example, among all the different locations, prefixes and suffixes, there is a part within a full library filename that can be taken as a *short library name* or label.

The function `dynfind` takes a list of short library names to locate a library using common search heuristics. For example, to load the Standard C Math library, one would either use the Microsoft Visual C Run-Time library labeled 'msvcrt' on Windows or the C Math library labeled 'm' or 'm.so.6' otherwise.

```
> mLib <- dynfind(c("msvcrt","m","m.so.6"))
```

`dynfind` also supports more exotic schemes, such as the Mac OS X Framework folders. Depending on the library, it is sometimes enough to have a single short filename - e.g. `"expat"` for the *Expat* library.

Internally, the dynamic linker interface of the OS is used via `.dynload` and symbols get resolved via `.dynsym`:

```
> sqrtAddr <- .dynsym(mLib, "sqrt")
```

[3]Possible C math library names are 'libm.so', 'libm.so.6' and 'MSVCRT.DLL' in locations such as '/lib', '/usr/lib', '/lib64', '/lib/sparcv9', '/usr/lib64', 'C:\WINDOWS\SYSTEM32' etc..

Although R already contains support for loading shared libraries and resolving of symbols, several issues have led to a reimplementation of this part:

- System paths are not considered when loading libraries via `dyn.load` of the package **base** but this is one part of the search heuristics.
- Automatic life-cycle management for loading and unloading of libraries is a desired goal. Unloading of libraries should be done automatically via finalizer code when no symbols are used anymore. External pointers resolved via `.dynsym` hold a reference to the loaded library. When all external pointers are garbage collected, the library handle is not referenced anymore and the finalizer can unload the library.

3.2 Wrapping C libraries

Functional R interfaces to foreign code can be defined with small R wrapper functions, which effectively delegates to `.dyncall`. Each function interface is parameterized by a target address and a matching call signature.

Since APIs often consist of hundreds of functions (see Table 1), `dynbind` can create and install a batch of function wrappers for a library with a single call by using a *library signature* that consists of concatenated function names and signatures separated by semicolons.

For example, to install wrappers to the C functions `sqrt`, `sin` and `cos` from the math library, one could use:

```
> dynbind( c("msvcrt","m","m.so.6"),
+ "sqrt(d)d;sin(d)d);cos(d)d;" )
```

The function call has the side-effect that three R wrapper functions are created and stored in an environment which defaults to the global environment. Let us review the `sin` wrapper (on the 64-bit Version of R running on Mac OS X 10.6):

```
> sin
function (...)
.dyncall.default(<pointer: 0x7fff81fd13f0>,
 "d)d)", ...)
```

The wrapper directly uses the address of the resolved `sin` symbol. In addition, the wrappers uses `.dyncall.default`, which is a concrete selector of a particular calling convention, as outlined below.

3.3 Calling Conventions

Calling conventions specify how arguments and return values are passed across sub-routines and functions at machine level. This information is vital for interfacing with the binary interface of C libraries. The package has support for multiple calling conventions. Calling conventions are controlled by `.dyncall` via the named argument `callmode` to specify a non-default calling convention. Most current OSs and platforms only have support for a single `"default"` calling convention at run-time.

An important exception is the Microsoft Windows platform on the 32-bit *i386* processor architecture: While the default C calling convention on *i386* is `"cdecl"` (which is the `"default"` on *i386*), system shared libraries from Microsoft such as 'KERNEL32.DLL', 'USER32.DLL' and the OpenGL library 'OPENGL32.DLL' use the `"stdcall"` calling convention. Only on this platform, the `callmode` argument has an effect and selects the calling convention to be used when working on Microsoft Windows 32-Bit. All other platforms currently ignore this argument.

3.4 Handling of C Types in R

C APIs often make use of high-level C `struct` and `union` types for exchanging information. Thus, to make interoperability work at that level the handling of C type information is addressed by the package.

Let us consider the following hypothetical example: A user-interface library has a function to set the 2D coordinates and dimension of a graphical output window. The coordinates are specified using a C `struct Rect` data type and the C function receives a pointer on that object:

```
void setWindowRect(struct Rect *pRect);
```

The structure type is defined as follows:

```
struct Rect {
  short          x, y;
  unsigned short w, h;
};
```

Before we can issue a call, we have to allocate an object of that size and initialize the fields with values encoded in C types, which are not part of R data types. The framework provides helper functions and objects to deal with C data types in R. Type information objects can be created with a description of the C aggregate structure. First, we create a type information object in R for the `struct Rect` C data type via `parseStructInfos` using a *structure type signature*.

```
> parserStructInfos("Rect{ssSS}x y w h;")
```

After registration, an R object named `Rect` is installed, which contains C type information that corresponds to `struct Rect`. The format of a *structure type signature* has the following pattern:

Struct-name `'{'` *Field-types* `'}'` *Field-names* `';'`

Field-types use the same type signature encoding as that of *call signatures* for argument and return types (Table 2). *Field-names* consist of a list of white-space separated names, labeling each field component.

An instance of a C type can be allocated via `new.struct`:

```
> r <- new.struct(Rect)
```

Finally, the extraction (`'$'`, `'['`) and replacement(`'$<-'`, `'[<-'`) operators can be used to access structure fields symbolically. During value transfer between R and C, automatic conversion of values with respect to the underlying C field type takes place.

```
> r$x <- -10 ; r$y <- -20 ; r$w <- 40 ; r$h <- 30
```

In this example, R `numeric` values are converted on the fly to `signed-` and `unsigned short` integers (usually 16-bit values). When the object gets printed on the prompt, a detailed picture of the data object is given:

```
> r
struct Rect {
 x: -10
 y: -20
 w:  40
 h:  30
}
```

At low-level, one can see that `r` is stored as an R `raw` vector object:

```
> r[]
[1] f6 ff ec ff 28 00 1e 00
attr(,"struct")
[1] "Rect"
```

To follow the example, we issue a foreign function call to `setRect` via `.dyncall` and pass in the `r` object, assuming the library is loaded and the symbol is resolved and stored in an external pointer object named `setWindowRectAddr`:

```
> .dyncall( setWindowRectAddr, "*<Rect>)v", r)
```

We make use of a typed pointer expression `'*<Rect>'` instead of the untyped pointer signature `'p'`, which would also work but does not prevent users from passing other objects that do not reference a `struct Rect` data object. Typed pointer expressions increase type-safety and use the pattern `'*<`*Type-Name*`>'`. The invocation will be rejected if the argument passed in is not of C type `Rect`. As `r` is tagged with an attribute `struct` that refers to `Rect`, the call will be issued.

Typed pointers can also occur as return types that - once the type information is available - permit the manipulation of returned objects in the same symbolic manner as above.

C `union` types are supported as well but use the `parseUnionInfos` function instead for registration and a slightly different signature format:

Union-name `'|'` *Field-types* `'}'` *Field-names* `';'`

The underlying low-level C type read- and write operations and conversions from R data types are performed by the functions `.pack` and `.unpack`. These can be used for various low-level operations as well, such as dereferencing of pointers on pointers.

R objects such as external pointers and atomic raw, integer and numeric vectors can be used as aggregate C types via the attribute `struct`. To *cast* a type in the style of C, one can use `as.struct`.

3.5 Wrapping R functions as C callbacks

Some C libraries, such as user-interface toolkits and I/O processing frameworks, use *callbacks* as part of their interface to enable registration and activation of user-supplied event handlers. A callback is a user-defined function that has a library-defined function type. Call-backs are usually registered via a registration function offered by the library interface and are activated later from within a library run-time context.

rdyncall has support for wrapping ordinary R functions as C callbacks via the function `new.callback`. Callback wrappers are defined by a *callback signature* and the user-supplied R function to be wrapped. *Callback signatures* look very similar to *call signatures* and should match the functional type of the underlying C callback. `new.callback` returns an external pointer that can be used as a low-level function pointer for the registration as a C callback. See Section *Parsing XML using Expat* below for applications of callback.

3.6 Foreign Library Interface

At the highest level, **rdyncall** provides the front-end function `dynport` to dynamically setup an interface to a C Application Programming Interface. This includes loading of the corresponding shared C library and resolving of symbols. During the binding process, a new R name space (Tierney, 2011) will be populated with thin R wrapper objects that represent abstractions to C

counter-parts such as functions, pointer-to-functions, type-information objects for C struct and union types and symbolic constant equivalents of C enums and macro defines. The mechanism aims to work across platforms, given that the corresponding shared libraries of a *DynPort* have been installed in a system standard location on the host.

An initial repository of *DynPorts* is available in the package that provides bindings for several popular C APIs, see Table 1 for examples of available bindings.

4 Sample Applications

We give two examples with different application contexts that demonstrate the direct usage of C APIs from within R through the **rdyncall** package. The R interface to C libraries looks very similar to the actual C API. For details on the usage of a particular C library, the programming manuals and documentation of the libraries should be consulted.

Before loading R bindings via `dynport`, the shared library should have been installed onto the system. Currently this is to be done manually and the installation method depends on the target OS (See the manual page about the 'rdyncall-demos' for details on this). While *OpenGL* is most often pre-installed on typical desktop-systems, *SDL* and *Expat* sometimes have to be installed explicitly.

4.1 OpenGL Programming in R

In the first example, we make use of the Simple DirectMedia Layer library (SDL) (Lantinga and et al, 2011) (Pendleton, 2003) (Wen, 2001) and the Open Graphics Library (OpenGL) (Board et al., 2005) to implement a portable multimedia application skeleton in R.

We first need to load bindings to SDL and OpenGL via dynports:

```
> dynport(SDL)
> dynport(GL)
```

Now we initialize the SDL library - in particular the video subsystem, and open a window surface with a dimension of $640x480$ in 32-bit color depths that has support for OpenGL rendering:

```
> SDL_Init(SDL_INIT_VIDEO)
> surface <- SDL_SetVideoMode(640,480,32,SDL_OPENGL)
```

Next, we implement the application loop which updates the display repeatedly and processes the event queue until a *quit* request is issued by the user via the window close button.

```
> mainloop <- function()
{
  ev <- new.struct(SDL_Event)
  quit <- FALSE
  while(!quit) {
    draw()
    while(SDL_PollEvent(ev)) {
      if (ev$type == SDL_QUIT) {
        quit <- TRUE
      }
    }
  }
}
```

SDL event processing is implemented by collecting events that occur in a queue. Once per update frame, typical SDL applications poll the queue by calling `SDL_PollEvent` with a pointer to a user-allocated buffer of C type `union SDL_Event`. Event records have a common type identifier which is set to `SDL_QUIT` when a quit event has occurred e.g. when users press a close button on a window.

Next, we implement our `draw` function making use of the OpenGL 1.1 API. We clear the background with a blue color and draw a light-green rectangle.

```
> draw <- function()
{
  glClearColor(0,0,1,0)
  glClear(GL_COLOR_BUFFER_BIT)
  glColor3f(0.5,1,0.5)
  glRectf(-0.5,-0.5,0.5,0.5)
  SDL_GL_SwapBuffers()
}
```

Now we can run the application mainloop.

```
> mainloop()
```

To stop the application, we hit the close button of the window. A similar example is also available via `demo(SDL)`. Here the `draw` function displays a rotating 3D cube depict in Figure 2.

`demo(randomfield)` gives a slightly more scientific application of OpenGL and R: Random fields of 512x512 size are generated via blending of 5000 texture mapped 2D gaussian kernels. The *frames per second* counter in the window title gives the number of matrices generated per second (see Figure 3). When clicking on the animation window, the current frame and matrix is passed to R and plotted. While several dozens of matrices are computed per second using OpenGL, it takes several seconds to plot a single matrix in R using `image()`.

4.2 Parsing XML using Expat

In the second example, we use the Expat XML Parser library (Clark, 2011) (Kim, 2001) to implement a stream-oriented XML parser suitable for very large documents.

The library, being very popular, is very likely to be already installed on many OS distributions - otherwise it is available from package repositories or can be built as a shared library from source.

In Expat, custom XML parsers are implemented by defining functions that are registered as callbacks to be invoked on events that occur during parsing, such as the start and end of XML tags. In our second example, we create a simple parser skeleton that prints the start and end tag names.

First we load R bindings for Expat via `dynport`.

```
> dynport(expat)
```

Next we create an abstract parser object via the C function `XML_ParserCreate` that receives one argument of type C string to specify a desired character encoding that overrides the document encoding declaration. We want to pass a null pointer (`NULL`) here. In the `.dyncall` FFI C null pointer values for pointer types are expressed via the R `NULL` value:

```
> p <- XML_ParserCreate(NULL)
```

The C interface for registration of start and end-tag event handler callbacks is given below:

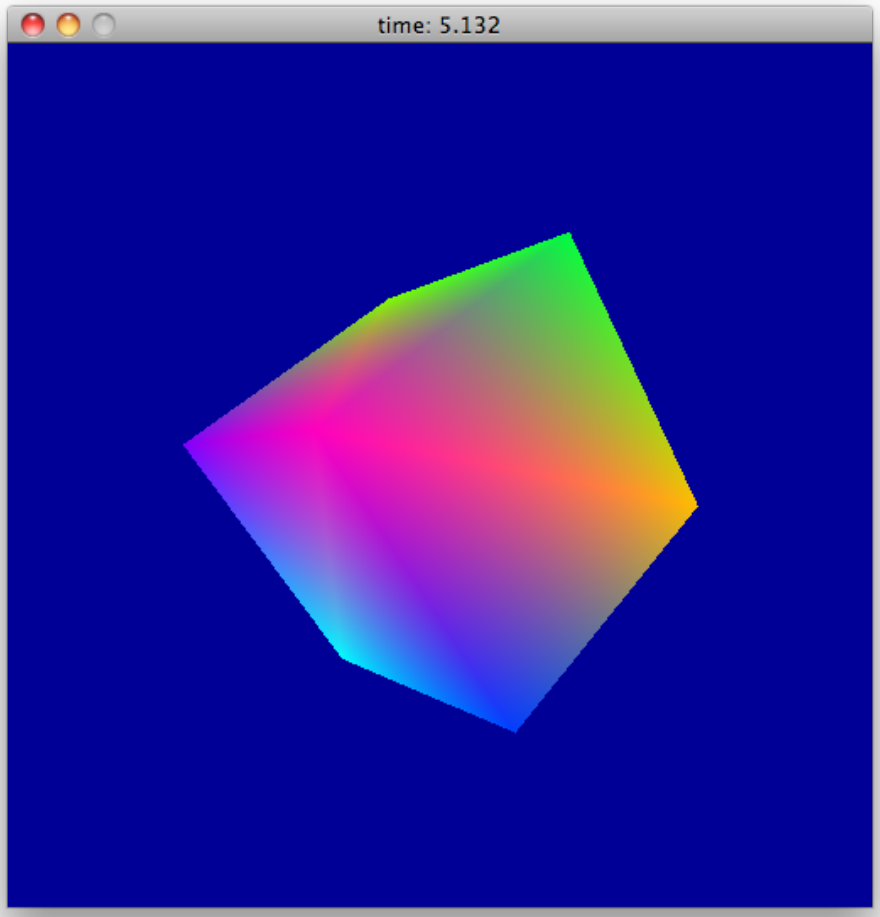

Figure 2: `demo(SDL)`

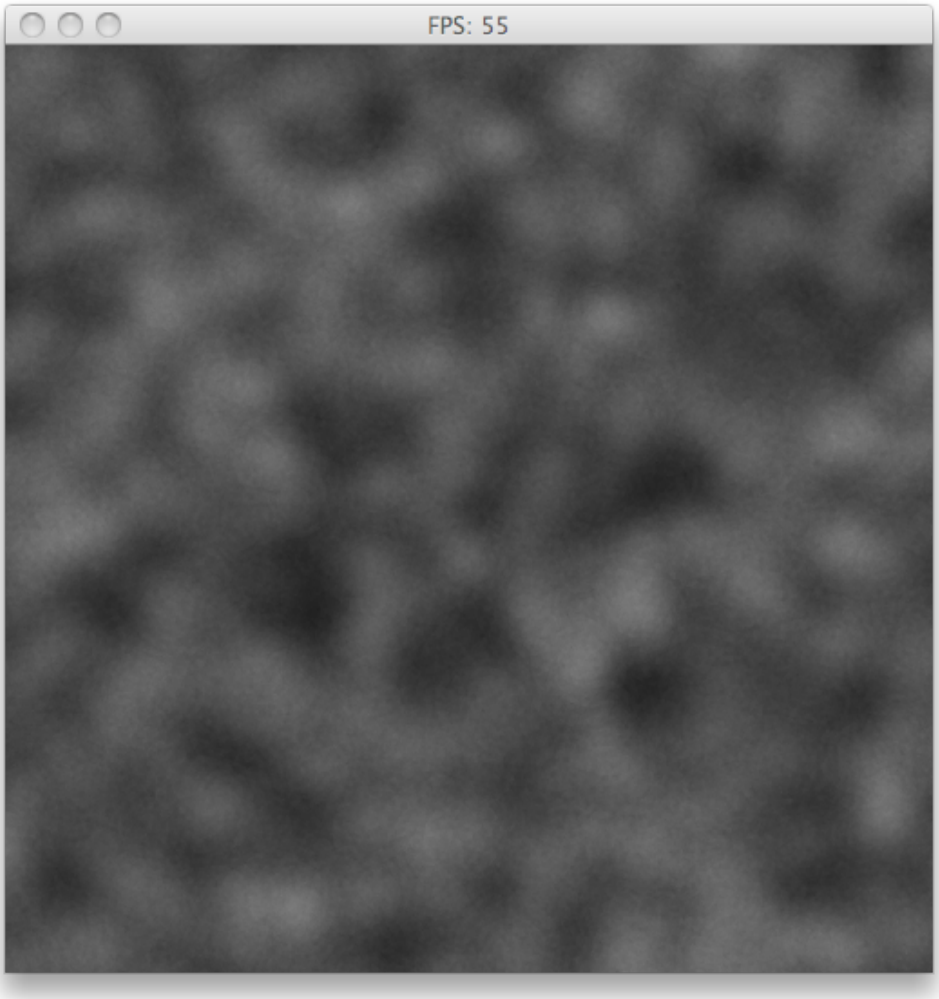

Figure 3: `demo(randomfield)`

```
/* Language C, from file expat.h: */
typedef void (*XML_StartElementHandler)
  (void *userData, const XML_Char *name,
   const XML_Char **atts);
typedef void (*XML_EndElementHandler)
  (void *userData, const XML_Char *name);
void XML_SetElementHandler(XML_Parser parser,
  XML_StartElementHandler start,
  XML_EndElementHandler end);
```

We implement the callbacks as R functions which print the event and tag name. They are wrapped as C callback pointers via `new.callback` using a matching *callback signature*. The second argument `name` of type C string in both callbacks, `XML_StartElementHandler` and `XML_EndElementHandler`, is of primnary interest ; this argument passes over the XML tag name. C strings are handled in a special way by the `.dyncall` FFI, because they have to be copied as R `character` objects. The special type signature `'Z'` is used to denote a C string type. The other arguments are simply denoted as untyped pointers using `'p'`:

```
> start <- new.callback("pZp)v",
  function(ignored1,tag,ignored2)
    cat("Start tag:", tag, "\n")
)
> end <- new.callback("pZ)v",
  function(ignored,tag)
    cat("Stop tag:", tag, "\n")
)
> XML_SetElementHandler(p, start, end)
```

To test the parser, we create a sample document stored in a `character` object named `text` and pass it to the parse function `XML_Parse`:

```
> text <- "<hello> <world> </world> </hello>"
> XML_Parse( p, text, nchar(text), 1)
```

The resulting output is given below:

```
Start tag: hello
Start tag: world
End tag: world
End tag: hello
```

Expat supports processing of very large XML documents in a chunk-based manner by calling `XML_Parse` several times, where the last argument is used as indicator for the final chunk of the document.

5 Architecture

The core implementation of the FFI, callbacks and loading of code are mainly based on the suite of libraries of the *DynCall* project (Adler and Philipp, 2011).

5.1 Dynamic calls

The FFI offered by **rdyncall** is based on the **dyncall** library, which provides an abstraction for making arbitrary machine-level calls with support for multiple calling conventions and most C argument- and return-types. [4]

For each processor architecture, the supported calling conventions are abstracted in a *Call Virtual Machine* (CallVM) object. The **dyncall** library offers a universal C interface that can be used from within scripting language interpreter contexts to build up a machine-level call in a structured manner.

A CallVM comprises a state machine and a call kernel. The state machine is implemented in C and keeps track of internal buffers for pre-loading argument values that get arranged for specific storage locations, such as stack or special register sets according to the processor architecture and the chosen calling conventions. The actual invocation of a foreign function call is conducted by the Call Kernel - a small piece of code that is implemented in Assembly and that provides a generic call facility for a particular calling convention. It prepares machine-level calls by copying data to registers and to the call stack according to the relevant calling convention, and finally executes the machine call to a target address.

From a scripting language interpreter perspective, the invocation of a foreign function call through the CallVM is conducted in three consecutive phases using the **dyncall** C API:

1. *Setup Phase:* The desired calling convention has to be chosen which, in most cases, is just the *default C* calling convention. However, more specialized and platform-specific calling conventions are available as well, in particular for the 32-Bit Windows OS.
2. *Argument Loading Phase:* Arguments are passed in a *left-to-right* order according to the declaration of the C/C++ function/method type declaration. Argument values are stored in buffers according to the processor architecture and selected calling convention.
3. *Call and Return-Value Receive Phase:* A return-type specific call function is chosen and the target address of the foreign code is passed, which gets called via the Call Kernel.

The architecture makes it straight-forward to implement a FFI for a dynamic language interpreter using a text parser for call signatures to drive the conversion of arguments and results. Similar FFIs with a text-based interface have been implemented for other language interpreters such as Ruby, Python and Lua. See the DynCall source repository (Adler and Philipp, 2011).

Both the C interface of dyncall and the signature format use the abstract C/C++ type system and give no indication about the effective size of a particular type. In experiments with several C APIs bound via **rdyncall** it turns out that the signatures do work cross-platform, if the fundamental type definitions of the C API do not change across platforms. In our tests and the presented examples, a wide range of C APIs have this property and type signatures are valid across platforms even when switching between 32- and 64-bit platforms.

5.2 Dynamic callbacks

The **dyncallback** library provides a framework to implement dynamic callbacks for language interpreters to wrap scripting functions as C function pointers. The framework offers a universal C interface for callback handler that is implemented once for a particular interpreter. The handler receives callback calls from C and forwards the call, including conversion of arguments, to a scripting function.

Handlers need to access machine-level arguments whose location can be on the stack, or in registers, depending on the processor architecture and calling convention. For that reason,

[4] *Inline* structure types are currently not fully supported.

the handler interface receives an abstract argument iterator that gives structured access to the arguments for passing over to the high-level language. Call-backs are created via an interface that pools a handler, language context, scripting function reference, callback type-information and other user data into a *single* native C function pointer, such that even very low-level C callbacks without user-supplied user-data can be addressed with the underlying technique. [5]

5.3 Portability and Stability

The requirements for porting the *DynCall* libraries to a new processor and/or platform are high: The calling conventions of a target processor platform have to be studied in detail, state machines have to be implemented in C and a small amount of code has to be written in Assembly which can be even non-portable across build tools on the same platform. Nevertheless **dyncall** (as of version 0.7) has support for many processor architectures such as Intel i386 (x86), AMD 64 (x64), PowerPC 32-bit,ARM (including Thumb extension), MIPS 32/64-bit and SPARC 32/64-bit including support for several platform-, processor- and compiler-specific calling conventions. **dyncallback** also supports major processor architectures such as Intel i386 (x86), AMD 64 (x64) and ARM and offers partial support for PowerPC 32-bit (support for Mac OS X/Darwin). Besides the processor architecture, the libraries are also explicitly ported and tested on various OS such as Linux, Mac OS X, Windows, the BSD family, Solaris, Haiku, Minix and Plan9. Support for embedded platforms such as Playstation Portable, Nintendo DS and iPhone OS is available as well.

DynCall contains a suite of testing tools for quality assurance. Included are test-case generators written in Lua and Python. Extreme call and callback scenarios are tested here to ensure correct passing of arguments and results. Before a release, the libraries and tests are built for a large set of architectures on **DynOS** (Philipp, 2011) - a batch-build system using full system emulators such as **QEmu**(Bellard, 2005) and **GXEmul**(Gavare, 2011) and various operating-system images to test release candidates and create pre-built binary releases of the library.

5.4 Text-based Signature Interfaces

A common property of the service interface presented here is the use of signature text formats. Signatures are used as descriptors for types, such as foreign function calls, callbacks and aggregate data types. The reasons that lead to the use of signatures as a high-level user-interface to interact with such services are given next:

1. Cross-language interface: Text format interfaces are available across high-level languages. Examples for cross-language text-based interfaces include regular expressions or `printf`-style formatted output descriptions.
2. Developer-friendly: The simplicity and compactness of the text-format enables developers to bridge with foreign code in interactive and rapid development sessions. C type signatures can be derived by hand with minimum effort: Fundamental types are encoded with a single character and the upper-case encodes an `unsigned` type.
3. Machine-neutral: In contrast to binary encoded type libraries, the data format is not affected by the endian model of the underlying platform.
4. Parser-friendly: The signature format can be used as driver code to perform foreign function calls. Implementations of parsers match the sequential design of **dyncall**'s CallVM and **dyncallback**'s argument iterator interface.

[5] This includes callbacks for sort routines of the Standard C library which lack user-data.

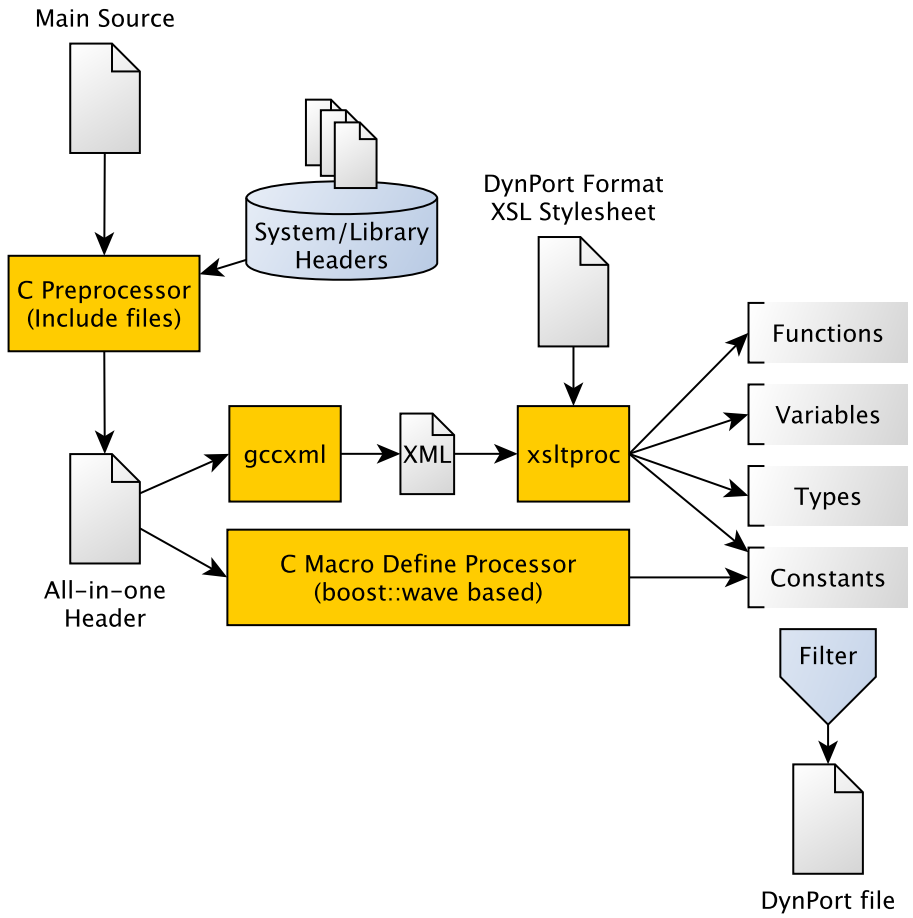

Figure 4: Tool-chain to create *DynPort* files from C headers

5.5 Creation of DynPort files

In this section we describe the tool-chain that creates the universal bindings called *DynPort*. The process described here is applied once on a build machine, the generated output is used later at run-time across platforms to drive the dynamic linkage and binding procedure. *DynPort* files can be created automatically from C header files using a tool-chain as depicted in Figure 4.

The tool-chain comprises several freely available components that are briefly described next: **GCC-XML** (King, 2011) is a modified version of the GCC compiler which translates C sources to XML document. **xsltproc**, distributed as part of the **libxslt** library (Veillard et al., 2011), is a XSLT processor that transforms XML documents to XML, text or binary formats according to style-sheets written in the *XSL Transformations* (Clark, 2001) language.

To extract library binding specifications, a main C source file is created that consists of one or more `#include` statements that reference library and/or system header files to process. The header files should have been previously installed on the build machine. In a preprocessing phase, the GNU C Macro Processor is used to process all `#include` statements using standard system search paths to create a concatenated *All-In-One* source file free of any `#include` statements. GCC-XML transforms C header declarations to XML. A XSL style-sheet implements the

transformation of XML to type signature formats using a XSLT processor. C Macro `#define` statements are handled separately by a custom C Preprocessor implemented in C++ using the boost wave library (Kaiser, 2011). An optional filter stage is used to include only elements with a certain pattern such as a common prefix usually found in many libraries e.g. '`SDL_`'. In a last step, the various fragments are assembled into a single text-file which represents the *DynPort* file. The overall build process is managed by *make* files and a repository of recipes has been setup to extend support for additional dynports and libraries in a structured and coordinated way.

6 Summary and Outlook

This paper introduces the **rdyncall** package (Version 0.7.3 on CRAN as of this writing) that contributes an improved Foreign Function Interface for R. The FFI facilitates *direct* invocation of foreign functions *without* the need to compile additional wrapper in C. Based on the FFI, a dynamic cross-platform linkage framework to wrap and access *whole* C interfaces of native libraries from R is discussed. Instead of *compiling* bindings for every library-and-language combination, R bindings of a library are created dynamically at run-time in a data-driven manner via *DynPort* files - a cross-platform universal type information format. C libraries are made accessible in R as though they were extension packages and the R interface looks very similar to that of C. This enables system-level programming in R and brings a new wave of possibilities for R developers such as using OpenGL directly in R across platforms as described in the example. An initial repository of *DynPort*s for standard cross-platform portable C libraries comes with the package.

The implementation is based on libraries from the *DynCall* project that implement non-trivial facilities such as an abstraction to machine-level function calls supporting multiple calling conventions and the handling of C callbacks from within scripting language interpreter environments. The libraries have been ported across major R platforms. Work is in progress to support missing architectures in **dyncallback** such as PowerPC System V 32-bit, PowerPC 64-bit, and, 32/64-bit MIPS and SPARC architectures. The handling of foreign aggregate data types, which is currently implemented in R and C, is planned to be reimplemented in portable C as part of *DynCall*, in cooperation with the developers of *BridJ*(Chafik, 2011). Currently, *DynPort* files are written as R scripts with inline text chunks created from the *DynPort* tool chain. For the Lua Programming Language (Ierusalimschy et al., 1996), a similar framework named **luadyncall** is in development using a language-neutral format for *DynPort* files. The need to install additional shared libraries still represents a hurdle for ordinary R users. We plan to find a common abstraction layer for installation systems, package managers and software distribution services across OS-distributions, and to integrate meta installation information into the *DynPort* file format.

The *DynPort* facility in **rdyncall** consitutes an initial step in building up an infrastructure between scripting languages and C libraries. Analogous to the way in which R users enjoy quick access to the large pool of R software managed by CRAN, we envision an archive network in which C library developers can distribute their work across languages, and users get quick access to the pool of C libraries from within scripting languages via automatic installation of precompiled components and using universal type information for cross-platform and cross-language dynamic bindings.

References

D. Adler and T. Philipp. DynCall Project. URL `http://dyncall.org`, May 2011.

F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical*

Conference, FREENIX Track, pages 41–46. USENIX, 2005. URL http://www.usenix.org/events/usenix05/tech/freenix/bellard.html.

O. A. R. Board, D. Shreiner, and et al. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2*. Addison Wesley, 2005.

O. Chafik. BridJ: Let Java & Scala call C, C++, Objective-C, C#... URL http://code.google.com/p/bridj/, May 2011.

J. Clark. XSL transformations (XSLT) version 1.1. W3C working draft, W3C, Aug. 2001. http://www.w3.org/TR/2001/WD-xslt11-20010824/.

J. Clark. The Expat XML Parser. URL http://expat.sourceforge.net/, May 2011.

A. Gavare. GXEmul: a framework for full-system computer architecture emulation. URL http://gxemul.sourceforge.net/, May 2011.

R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua—an extensible extension language. *Software—Practice and Experience*, 26(6):635–652, June 1996.

H. Kaiser. Wave V2.0 - Boost C++ Libraries. URL http://www.boost.org/doc/libs/release/libs/wave/index.html, May 2011.

E. E. Kim. A Triumph of Simplicity: James Clark on Markup Languages and XML. *Dr. Dobb's Journal of Software Tools*, 26(7):56, 58–60, July 2001. ISSN 1044-789X. URL http://www.ddj.com/.

B. King. GCC-XML. URL http://www.gccxml.org, May 2011.

S. Lantinga and et al. libSDL: Simple DirectMedia Layer. URL http://www.libsdl.org/, May 2011.

J. K. Ousterhout. Scripting: Higher Level Programming for the 21st Century, May 1997. URL http://www.sunlabs.com/people/john.ousterhout/scripting.ps. White Paper.

B. Pendleton. Game Programming with the Simple DirectMedia Layer (SDL). *Linux Journal*, 110:42, 44, 46, 48, June 2003. ISSN 1075-3583.

T. Philipp. DynOS Project. URL http://dyncall.org/dynos, May 2011.

R Development Core Team. *Writing R Extesions*. R Foundation for Statistical Computing, Vienna, Austria, 2010. URL http://www.R-project.org. ISBN 3-900051-11-9.

Ross Ihaka and Robert Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996. URL http://www.amstat.org/publications/jcgs/.

L. Tierney. A Simple Implementation of Name Spaces for R. URL http://www.stat.uiowa.edu/~luke/R/namespaces/morenames.pdf, May 2011.

D. Veillard, B. Reese, and et al. The XSLT C library for GNOME. URL http://xmlsoft.org/XSLT/, May 2011.

H. Wen. SDL: The DirectX Alternative. URL http://linuxdevcenter.com/pub/a/linux/2001/09/21/sdl.html, September 2001.