*Journal of Statistical Software*

*December 2010, Volume 37, Issue 8.* *http://www.jstatsoft.org/*

# RGtk2: A Graphical User Interface Toolkit for R

**Michael Lawrence**
Fred Hutchinson Cancer Research Center

**Duncan Temple Lang**
University of California, Davis

**Abstract**

Graphical user interfaces (GUIs) are growing in popularity as a complement or alternative to the traditional command line interfaces to R. **RGtk2** is an R package for creating GUIs in R. The package provides programmatic access to **GTK+** 2.0, an open-source GUI toolkit written in C. To construct a GUI, the R programmer calls **RGtk2** functions that map to functions in the underlying **GTK+** library. This paper introduces the basic concepts underlying **GTK+** and explains how to use **RGtk2** to construct GUIs from R. The tutorial is based on simple and pratical programming examples. We also provide more complex examples illustrating the advanced features of the package. The design of the **RGtk2** API and the low-level interface from R to **GTK+** are discussed at length. We compare **RGtk2** to alternative GUI toolkits for R.

*Keywords*: graphical user interface, GUI, R, **GTK+**.

## 1. Introduction

An interface, in the most general sense, is the boundary across which two entities communicate. In most cases, the communication is bidirectional, involving input and output from both of the interfaced entities. In computing, there are two general types of interfaces: machine interfaces and user interfaces (Unwin and Hofmann 1999). A machine interface does not involve humans, while a user interface is between a human and a machine. This paper discusses a machine interface between two software components, the R platform for statistical computing (R Development Core Team 2010) and **GTK+**, a library for constructing graphical user interfaces (**GTK+** Development Team 2010; Krause 2007).

Two common types of user interface in statistical computing are the command line interface (CLI) and the graphical user interface (GUI). The usual CLI consists of a textual console where the user types a sequence of commands at a prompt. The R console is an example of a CLI. A GUI is the primary means of interacting with desktops, like Windows and Mac OS, and statistical software like **JMP** (SAS Institute Inc. 2007). These interfaces are based on the

WIMP (window, icon, menu, and pointer) paradigm (Penners 2005). WIMP was developed at Xerox PARC in the 1970's and was popularized by the Apple Macintosh. On a WIMP desktop, application GUIs are contained within windows, and resources, such as documents, are represented by graphical icons. User controls are packed into hierarchical drop-down menus, buttons, sliders, etc. The user manipulates the windows, icons and menus with a pointer device, such as a mouse. The windows, icons, and menus, as well as other graphical controls such as buttons, sliders and text fields, have come to be known as *widgets*. The graphical event-driven, non-procedural nature and overall complexity of widgets makes their implementation a non-trivial task. To alleviate the burden on the application programmer, reusable widgets are collected into *widget toolkits*.

There is often debate over the relative merits of a CLI and a GUI lacking a console. The comparison largely depends on the skills and needs of the user (Unwin and Hofmann 1999). Effective use of a CLI requires the user to be proficient in the command language understood by the interface. For example, with a CLI, R users need to understand the R language. Learning a computer language often demands a significant commitment of time and energy; however, given a small amount of knowledge, one can use the language to perform arbitrary, rich tasks. A graphical interface is much less general and restrictive, but typically makes performing a specific task easier. It does this two different ways: a) stream-lining the steps involved in the task by providing a constrained context, and b) removing the need to remember function names and syntax. Different users benefit from the two different interfaces for different tasks. And there is little doubt that for occasional users of a language and for users focused a specific task, a well-designed GUI is easier to learn and more accessible than a general purpose programming language.

Considering the widespread use and popular appeal of the R platform and the rich set of state-of-the-art statistical methodology it provides, it is desirable to try to make these available to a broader set of users by simplifying the knowledge needed to use such methods. The CLI has always been the most popular interface to R as it is the generic interface provided on all platforms and there has been much less focus in the R community on providing graphical interfaces for specific tasks. On some platforms, a CLI is a component of a larger GUI with menus containing various utilities for working with R. Examples of CLI-based R GUIs include the official Windows and Mac OS X GUIs, as well as the cross-platform Java GUI for R (**JGR**, Helbig *et al.* 2005). Although these interfaces are GUIs, they are still very much in essence CLIs, in that the primary mode of interacting with R is the same. Thus, these GUIs appeal mostly to the power users of R. A separate set of GUIs targets the second group of users, those learning the R language. Since this group includes many students, these GUIs are often designed to teach general statistical concepts in addition to R. A CLI component is usually present in the interface, though it is deemphasized by the surrounding GUI, which is analogous to a set of "training wheels" on a bicycle. Examples of these GUIs include Poor Man's GUI (**pmg**, Verzani 2010) and R Commander (Fox 2005). The third group of users, those who only require R for certain tasks and do not wish to learn the language, are targeted by task-specific GUIs. These interfaces usually do not contain a command line, as the limited scope of the task does not require it. If a task-specific GUI fits a task particularly well, it may even appeal to an experienced user. There are many examples of task-specific GUIs in R, including **explorase** (Cook *et al.* 2008), **limmaGUI** (Smyth 2005), and **Rattle** (Williams 2010).

The task-specific GUIs, as well as more general R GUIs, are often implemented in the R language. The main advantage to writing a GUI in R is direct access to its statistical analysis functionality. The extensible nature of the R language and its support for rapid prototyping particularly faciliate the construction of task-specific GUIs. Building a GUI in R, as in any language, is made easier through the use of a widget toolkit. The **tcltk** package (Dalgaard 2001, 2002), which provides access to Tcl/Tk (Ousterhout 1994; Welch 2003), is the most often used GUI toolkit for R. Others include **RGtk** (Temple Lang 2001-2005), based on **GTK+**; **RwxWidgets** (Temple Lang 2008b), based on wxWidgets (Smart *et al.* 2005); and **gWidgets** (Verzani 2009), a simplified, common interface to several toolkits, including **GTK+**, Tcl/Tk and Java **Swing**. There are also packages for embedding R graphics in custom interfaces, such as **cairoDevice** (Lawrence 2009) and (the now defunct) **gtkDevice** (Drake *et al.* 2005) for **GTK+** and **tkrplot** (Tierney 2008) for Tcl/Tk.

**RGtk2** is a GUI toolkit for R derived from the **RGtk** package. Like **RGtk**, **RGtk2** provides programmatic access to **GTK+**, a cross-platform (Windows, Mac, and Linux) widget toolkit. The letters *GTK* stand for the *GIMP ToolKit*, with the word *GIMP* recording the origin of the library as part of the GNU Image Manipulation Program. **GTK+** is written in C, which facilitates access from languages like R that are also implemented in C. It is licensed under the *Lesser GNU Public License* (LGPL). **GTK+** provides the same widgets on every platform, though it can be customized to emulate platform-specific look and feel. The original **RGtk** is bound to the previous generation of **GTK+**, version 1.2. **RGtk2** is based on **GTK+ 2.0**, the current generation. Henceforth, this paper will only refer to **RGtk2**, although many of the fundamental features of **RGtk2** are inherited from **RGtk**. The package is available from the Comprehensive R Archive Network (CRAN) at `http://CRAN.R-project.org/package=RGtk2`.

We continue with the fundamentals of the **GTK+** GUI and the **RGtk2** package. This is followed by a tutorial, including examples, on using **RGtk2** to construct basic to intermediate GUIs. The paper then moves into a more technical domain, introducing the advanced features of the interface, including the creation of new types of widgets. We then present a technical description of the design and generation of the interface, which is followed by a discussion of more general binding issues. Next, we compare **RGtk2** to existing GUI toolkits in R. We conclude by mentioning some applications of **RGtk2** and explore directions for future development.

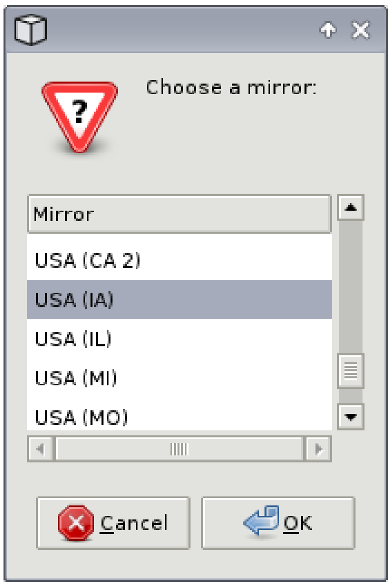
## 2. Fundamentals

This section begins with an introduction to the basic widgets and elements of the of the **GTK+** library. We then turn our attention to the **RGtk2** interface to **GTK+**, explaining how to create and manipulate widgets and how to respond to user input. The section concludes by introducing widget layout, the process of determining the size and position of each widget on the screen.

### 2.1. GTK+ widgets

#### *Widget type hierarchy*

The left panel of Figure 1 shows a **GTK+** GUI that allows the user to select a CRAN mirror for downloading R packages. This GUI is likely familiar to many R users, since a similar



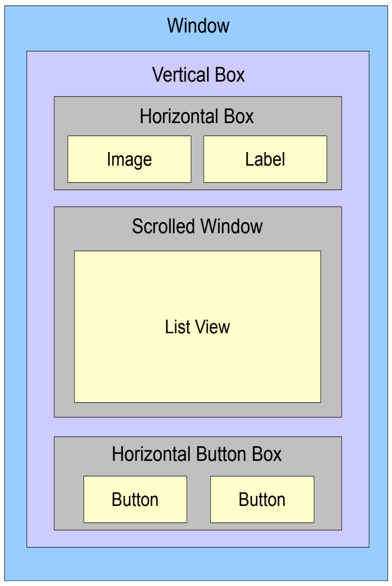


Figure 1: A dialog for selecting a CRAN mirror constructed using the **RGtk2** package. The screenshot of the dialog is shown on the left. The user selects a mirror from the list and clicks the `OK` button to confirm the choice. In the image on the right, each rectangle corresponds to a widget in the GUI. The window is at the top-level, and each of the other widgets is geometrically contained within its parent. Many of the container widgets are invisible in the screenshot.

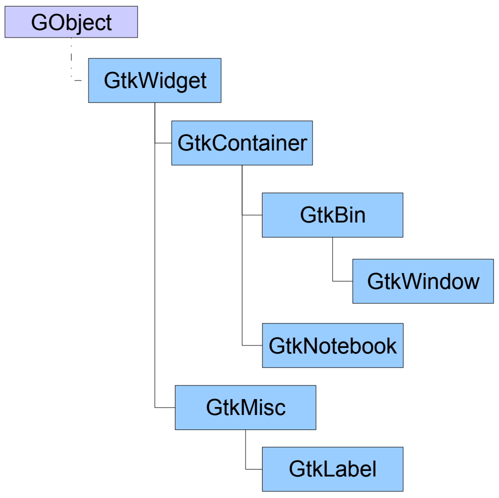


Figure 2: A small portion of the GTK+ class hierarchy. All widgets are derived from the `GtkWidget` class, which is derived, indirectly, from the `GObject` base class.

interface is present in the official Windows and Mac OS X R GUIs, among others. There are several different types of widgets in the CRAN mirrors GUI. A text label instructs the user to choose a mirror. A list control/widget contains the names of the available mirrors, and there are buttons for confirming or canceling the choice. The interface is enclosed by another type of widget, a window.

All of these widget types have functionality in common. For example, they are all drawn on the screen in a consistent style. To formalize this relationship and to simplify implementation by sharing code between widgets, **GTK+** defines an inheritance hierarchy for its widget types, or classes. A small portion of the **GTK+** class hierarchy is shown in Figure 2. For specifying the hierarchy, **GTK+** relies on **GObject**, a C library that implements a class-based, single-inheritance object-oriented system. Each type of **GTK+** widget is a **GObject** class that inherits from the base `GtkWidget` class which provides the general characteristics shared by all widget classes, e.g., properties giving the location, color; methods for hiding, showing and painting the widget. A **GObject** class encapsulates behaviors that all instances of the class share. Each class has a single parent from which it inherits the behaviors of its ancestors. A class can override some specific inherited behaviors. A more detailed and technical explanation of **GObject** is available in Section 5.2.

*Widget container hierarchy*

There is another tree hierarchy that is orthogonal to the class inheritance hierarchy. This hierarchy involves widget instances rather than widget classes. Each widget instance has a single parent instance in which it is contained, except for a top-level window which has no parent and serves as the root of the tree. Child widgets are contained within the rectangular region of their parents. In Figure 1, for example, the label, list of mirrors, and buttons are all contained within the top-level window, meaning that the window is the common ancestor of the other widgets. The right panel of Figure 1 shows, in a simplified way, the two dimensional nesting of the widgets in the mirror selection example. Widgets that can contain other widgets are called *containers* and their classes are derived from the `GtkContainer` class. Windows and tabbed notebooks are examples of containers. Combining primitive widgets like labels and icons within containers leads to more complex displays, such as menus, tool bars and even buttons which contain labels to display the text. A container is responsible for allocating its space to its children. This process is called layout management and is described in Section 2.3.

## 2.2. GTK+ widgets in R

**RGtk2** provides an Application Programming Interface (API) to the **GTK+** library. A programmer uses an API to create an application based on functions implemented within a separate module. It is a contract that specifies in detail the functionality available to a programmer without specifying how that functionality is implemented.

As with other user interfaces, an API should be consistent and efficient to use. As an R package, **RGtk2** primarily aims to be consistent with R conventions. This means hiding aspects of the **GTK+** API that are foreign to R, such as explicit memory management. A secondary concern is consistency with the underlying **GTK+** API. The developers of **GTK+** have invested a significant amount of thought into its design. Thus, **RGtk2** endeavors to interface R to the virtual entirety of **GTK+**, without leaving any gaps that may be unanticipated by the user. The only omissions are those that would violate consistency with R. For example,

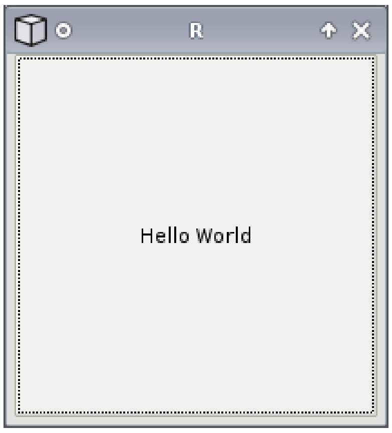


Figure 3: "Hello World" in GTK+. A window containing a single button displaying a label with the text `Hello World`.

functions related to explicit memory management were excluded, as memory in R is managed by a garbage collector. Array length parameters are also excluded, as the length of a vector is always known in R. The **RGtk2** API has also been designed for ease/efficiency of use. Towards this end, it specifies a default value for a function parameter whenever sensible and uses a special object-oriented syntax, as introduced by the **SJava** package (Temple Lang 2006b).

To demonstrate the basic syntax and features of the **RGtk2** API, we will construct a simple "Hello World" GUI, shown in Figure 3.

We will gradually progress from this trivial GUI to the aforementioned CRAN mirrors GUI and beyond. The first step is to create a top-level window to contain our GUI. Creating an instance of a **GTK+** widget requires calling a single R function, known as a constructor. The constructor for a class has the same name as the class, except the first character is lowercase. The following statement constructs an instance of the `GtkWindow` class:

```
R> window <- gtkWindow("toplevel", show = FALSE)
```

The first argument to the constructor for `GtkWindow` corresponds to the type of the top-level window. The set of possible window types is specified by what in C is known as an *enumeration*. Since enumerations are foreign to R, **RGtk2** accepts string representations of enumeration values, like `"toplevel"`. For every **GTK+** enumeration, **RGtk2** provides an R vector that maps the nicknames to the underlying numeric values. In the above case, the vector is named `GtkWindowType`. The expression `names(GtkWindowType)` returns the names of the possible values of the `GtkWindowType` enumeration, and the same applies to all other enumerations. It is rarely necessary to explicitly use the enumeration vectors; specifying the nickname will work in most cases, including all method invocations and is preferable as it is easier for human readers to comprehend.

The `show` argument is the last argument for every widget constructor. It indicates whether the widget should be made visible immediately after construction. The default value of `show` is `TRUE`. In this case we want to defer showing the window until after we finish constructing our simple GUI.

The next steps are to create a "Hello World" button and to place the button in the window that we have already created. This depends on an understanding of how of one programmatically manipulates widgets. Each widget class defines an API consisting of methods, properties, fields and signals. Methods are functions that take an instance of their class as the first argument and are used to instruct the widget to perform an action. Properties and fields store the public state of a widget. Examples of properties include the title of a window, the label on a button, and whether a widget has the keyboard focus. Signals are emitted as a result of events, such as user interaction with a widget. By attaching an R handler function to a widget's signal, we can perform an action in response to all user inputs that generate that signal. We explain how one can interface R functions with each of these in the following sections as we continue with our "Hello World" example.

### *Invoking methods*

Methods are functions that operate on widgets inheriting from a particular class. The **RGtk2** function for each **GTK+** method is named according to the *classNameMethodName* pattern. For example, to add a child to a container, we need to invoke the `add` method on the `GtkContainer` class. The corresponding function name would be `gtkContainerAdd`. However, this introduces an inefficiency in that the user needs to remember the class to which a method belongs. To circumvent this problem, we introduce a syntax that is similar to that found in various object-oriented languages. The widget variable is given first, followed by the `$` operator, then the method name and its arguments. This syntax for calling `gtkContainerAdd` is demonstrated below as we add a button with the label `Hello World` to our window. The third statement calls `gtkWindowSetDefaultSize` to specify our desired size for the window when it is first shown. The code for these method invocations is below:

```
R> button <- gtkButton("Hello World")
R> window$add(button)
R> window$setDefaultSize(200, 200)
```

Each method belongs to a separate class, but the syntax frees the user from the need to remember the exact classes and also saves some typing as the `$` operator finds the most specific/appropriate method based on the class inheritance of the widget. Note that we use the lower case form of the first letter when using the `$` syntax, but the upper case form in the `classNameMethodName` function name. The `$` acts as a word separator and we use lower case at the beginning of new words.

### *Accessing properties and fields*

Properties are self-describing elements that store the state of an aspect of a widget. Examples of properties include the title of a window, whether a checkbox is checked, and the length in characters of a text entry box. The R subset function `[` may be used to get the value of a widget property by name. Below we access the value of the `visible` property of our window:

```
R> window["visible"]

[1] FALSE
```

We find that the value is `FALSE`, since we specified it not to be shown at construction and have not made it visible since then.

**GTK+** properties may be set, given that they are writable, using the regular R assignment operator (`<-` or `=`). This is actually implemented via the `[<-` method for **GTK+** widgets in **RGtk2**. The example below makes the window created above visible, using both property-setting methods, the second corresponding to a call to `gtkWidgetShow`, which is more conventional:

```
R> window["visible"] <- TRUE
R> window$show()
```

For convenience, one might desire to set multiple properties with a single statement. This is possible using the `gObjectSet` method, which behaves similarly to the R `options` function, in that the argument name indicates the property to set to the argument value. In the single statement below, we set the window icon to the **RGtk** logo image and set the title to `"Hello World 1.0"`:

```
R> image <- gdkPixbuf(filename = imagefile("rgtk-logo.gif"))[[1]]
R> window$set(icon = image, title = "Hello World 1.0")
```

The `imagefile` function retrieves an image from the **RGtk2** installation. `gdkPixbuf` returns a list, where the first element is a `GdkPixbuf`, an image object, and the second is a description of an error encountered when reading the file or `NULL` if the operation was successful. Here we assume that there is no error.

In rare cases, it is necessary to access a field in the widget data structure. Fields are different from properties in several ways. Most importantly, it is never possible to set the value of a field. The user can retrieve the value of a field using the `[[` function. For example, now that our window has been shown, it has been allocated a rectangle on the screen. This is stored in the `allocation` field of `GtkWidget`. It returns a list representing a `GtkAllocation` with elements `x`, `y`, `width` and `height`, as in the code segment below:

```
R> window[["allocation"]]

$x
[1] 0

$y
[1] 0

$width
[1] 200

$height
[1] 200

attr(,"class")
[1] "GtkAllocation"
```

### *Handling signals/events*

Once a GUI is displayed on the screen, the user is generally free to interact with it. Examples of user actions include clicking on buttons, dragging a slider and typing text into an entry box. In the CRAN mirrors example, possible user actions include selecting a mirror in the list, clicking the `OK` or `Cancel` buttons and pressing a keyboard shortcut, such as `Alt-O` for `OK`. An application may wish to respond in a certain way to one or more of such actions. The CRAN mirrors application, for example, should respond to an `OK` response by saving the chosen mirror in the session options.

So far, we have created and manipulated widgets by calling a list of procedures in a fixed order. This is convenient as long as the application is ignoring the user. Listening to the user would require a loop which continuously checks for user input. It is not desirable to implement such a loop for every application, so **GTK+** provides one for all GUI applications to use within the same R session. When an application initializes the **GTK+** event processing loop, there is an *inversion of control.* The application no longer has primary control of its flow; instead, **GTK+** asynchronously informs the application of events through the invocation of functions provided by the application to handle a specific type of event. These handlers are known as *callbacks*, because **GTK+** is calling back into the application.

**GTK+** widgets represent event types as signals. One or more callbacks can be connected to a signal for each widget instance. When the event corresponding to the signal occurs, the signal is emitted and the callbacks are executed in an order depending on how they were connected. In order to execute R code in response to a user action on a widget, we connect an R function to the appropriate signal on the widget. The `gSignalConnect` function performs this connection. The following code will make our "Hello World" example from above more interactive:

```
R> gSignalConnect(button, "clicked", function(widget) print("Hello world!"))
```

The call to `gSignalConnect` will cause `"Hello world!"` to be printed upon emission of the `clicked` signal from the button in our window. The `clicked` signal is emitted when the user clicks the button with a pointer device or activates the button with a keyboard shortcut.

### *Widget documentation*

Documentation for widgets is available using the conventional R `help` command. It is derived from the documentation of **GTK+** itself. To see the methods, properties, fields, and signals available for a particular class, the user should access the help topic matching the class name. For example, to read the documentation on `GtkWindow` we enter:

```
R> help("GtkWindow")
```

Similarly, the detailed help for a specific method is stored under the full name of the function. For example, to learn about the `add` method on `GtkContainer`, we enter:

```
R> help("gtkContainerAdd")
```

## 2.3. Widget layout

In our "Hello World" example, we added only a single widget, a button, to the top-level window. In contrast, the CRAN mirrors window contains multiple widgets, which introduces the problem of appropriately allocating the space in a window to each of its descendents in the container hierarchy. This problem is often called *layout management*. Laying out a GUI requires specifying the position and size of each widget below the top-level window. The simplest type of layout management is static; the position and size of each widget are fixed to specific values. This is possible with **GTK+**, but it often yields undesirable results. A GUI is interactive and changes in response to user input. The quality of a fixed layout tends to decrease with certain events, such as the user resizing the window, a widget changing its size requirement, or the application adding or removing widgets. For this reason, most layout management is dynamic.

In **GTK+**, containers are responsible for the layout of their children. The right panel in Figure 1 shows how the nesting of layout containers results in the CRAN mirrors GUI shown in Figure 1. The example employs several important types of **GTK+** layout containers.

First, there is the top-level `GtkWindow` that is derived from `GtkBin`, which in turn derives from `GtkContainer`. A `GtkBin` holds only a single child, and `GtkWindow` simply fills all of its allocated space with its child.

The most commonly used container for holding multiple children is the general `GtkBox` class, which stacks its children in a specified order and in a single direction, vertical or horizontal. The children of a `GtkBox` always fill the space allocated to the box in the direction orthogonal to that of the stacking, e.g., fill the available width when stacked vertically on top of each other.

The `GtkBox` class is abstract (or virtual), meaning that one cannot create instances of it. Instead, we instantiate one of its non-abstract subclassses. For example, in the CRAN mirror GUI, a vertical box, `GtkVBox`, stacks the label above the list, and a horizontal button box, `GtkHButtonBox`, arranges the two buttons. `GtkVBox` and its horizontal analog `GtkHBox` are general layout containers, while the button boxes `GtkVButtonBox` and `GtkHButtonBox` offer facilities specific to the layout of sets of buttons.

Here we will explain and demonstrate the use of `GtkHBox`, the general horizontal box layout container. `GtkVBox` can be used exactly the same way; only the direction of stacking is different. Figure 4 illustrates a sampling of the possible layouts that are possible with a `GtkHBox`.

The code for some of these layouts is presented here. We begin by creating a `GtkHBox` widget. We pass `TRUE` for the first parameter, `homogeneous`. This means that the horizontal allocation of the box will be evenly distributed between the children. The second parameter directs the box to leave 5 pixels of space between each child. The following code constructs the `GtkHBox`:

```
R> box <- gtkHBox(TRUE, 5)
```

The equal distribution of available space is strictly enforced; the minimum size requirement of a homogeneous box is set such that the box always satisfies this assertion, as well as the minimum size requirements of its children.

The `gtkBoxPackStart` and `gtkBoxPackEnd` methods pack a widget into a box with left and right justification (top and bottom for a `GtkVBox`), respectively. For this explanation, we

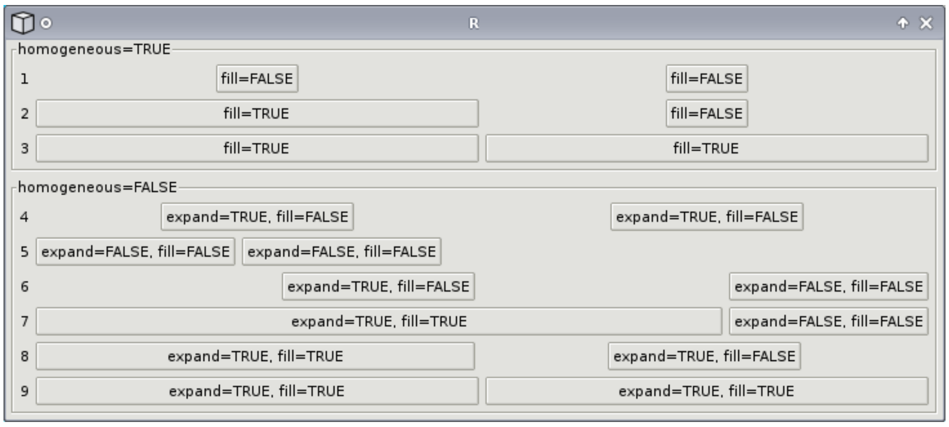

R
homogeneous=TRUE
1 fill=FALSE fill=FALSE
2 fill=TRUE fill=FALSE
3 fill=TRUE fill=TRUE
homogeneous=FALSE
4 expand=TRUE, fill=FALSE expand=TRUE, fill=FALSE
5 expand=FALSE, fill=FALSE expand=FALSE, fill=FALSE
6 expand=TRUE, fill=FALSE expand=FALSE, fill=FALSE
7 expand=TRUE, fill=TRUE expand=FALSE, fill=FALSE
8 expand=TRUE, fill=TRUE expand=TRUE, fill=FALSE
9 expand=TRUE, fill=TRUE expand=TRUE, fill=TRUE

Figure 4: A screenshot demonstrating the effect of packing two buttons into `GtkHBox` instances using the `gtkBoxPackStart` method with different combinations of the `expand` and `fill` settings. The effect of the `homogeneous` spacing setting on the `GtkHBox` is also shown.

restrict ourselves to `gtkBoxPackStart`, since `gtkBoxPackEnd` works the same except for the justification. Below, we pack two buttons, `button_a` and `button_b` using left justification:

```
R> button_a <- gtkButton("Button A")
R> button_b <- gtkButton("Button B")
R> box$packStart(button_a, fill = FALSE)
R> box$packStart(button_b, fill = FALSE)
```

First, `button_a` is packed against the left side of the box, and then we pack `button_b` against the right side of `button_a`. The space distribution is homogeneous, but the extra space for each widget is not filled. This results in the first row in Figure 4.

Making the space available to a child does not mean that the child will fill it. That depends on the minimum size requirement of the child, as well as the value of the `fill` parameter passed to `gtkBoxPackStart`. When a widget is packed with the `fill` parameter set to `TRUE`, the widget is sized to consume the available space. This results in rows 2 and 3 in Figure 4.

In many cases, it is desirable to give children unequal amounts of available space, as in rows 4–9 in Figure 4. This is evident in the CRAN mirrors dialog, where the mirror list is given more space than the `Please choose a mirror` label. To create an inhomogeneously spaced `GtkHBox`, we pass `FALSE` as the first argument to the constructor, as in the following code:

```
R> box <- gtkHBox(FALSE, 5)
```

An inhomongeneous layout is freed of the restriction that all widgets must be given the same amount of available space; it only needs to ensure that each child has enough space to meet its minimum size requirement. After satisfying this constraint, a box is often left with extra space. The programmer may control the distribution of this extra space through the `expand` parameter to `gtkBoxPackStart`. When a widget is packed with `expand` set to `TRUE`, we will call the widget an *expanding* widget. All expanding widgets in a box are given an equal portion of the entirety of the extra space. If no widgets in a box are expanding, as in row 5 of Figure 4, the extra space is left undistributed. It is common to mix expanding and non-expanding widgets in the same box. For example, in the CRAN mirrors dialog, the box first ensures that the mirror list and the label above it are given enough space to satisfy their minimum requirement. Then, since the mirror list is expanding, all of the extra space is made available to it, while the label is left only with its minimum requirement (i.e., enough space to show its text). Another example is given below, where `button_a` is expanding, while `button_b` is not:

```
R> box$packStart(button_a, expand = TRUE, fill = FALSE)
R> box$packStart(button_b, expand = FALSE, fill = FALSE)
```

The result is shown in row 6 of Figure 4. The figure contains several other permutations of the `homogeneous`, `expand` and `fill` settings.

**GTK+** contains many types of layout containers besides boxes, including a grid layout (`GtkTable`), a user-adjustable split pane (`GtkHPaned` and `GtkVPaned`), and a tabbed notebook (`GtkNotebook`). More types of layout containers will be demonstrated later in the tutorial.

## 3. Basic GUI construction

Thus far, we have reviewed the fundamentals of **GTK+**, working with **GTK+** widgets from R, and widget layout management. In this section, we will build on this foundation to create some basic but potentially useful GUIs.

Constructing a GUI may be conceptually divided into two basic steps. First, one must create the individual widgets, specify their properties, and organize them into containers. This defines the physical aspect of the GUI: the appearance of the widgets and their spatial organization. The second step defines the behavior or the logical aspect of the interface. It involves registering handlers for signals that are emitted by the widgets, for example in response to a user pressing a button. The signal handlers encapsulate the logic beneath the interface. In this section, we will demonstrate these two steps and show how their integration results in functional GUIs.

### 3.1. A dialog with the user

A user interface is the conduit for a conversation between the machine and the user. This conversation may be broken down into a series of exchanges called *dialogs*. An application often needs to make a specific request for user input, such as the desired CRAN mirror. This type of dialog is initiated by the machine posing a question to the user. The machine then waits for the user to respond. Usually, the application is unable to continue until receiving the user response, so the rest of the GUI is blocked until the dialog is concluded. This is called a *modal* dialog. A dialog is described as *non-modal* when the user can continue to perform other tasks even when the dialog is displayed.

**GTK+** explicitly supports modal and non-modal requests for user input with a dialog widget, a top-level window that emits the `response` signal when the user has responded to the query. All dialogs in **GTK+** are derived from the `GtkDialog` class. The CRAN mirrors GUI is

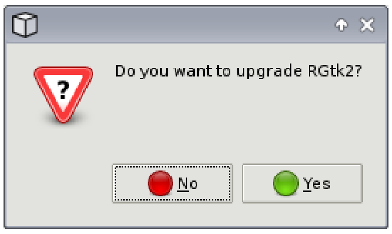


Figure 5: A screenshot of a message dialog requesting a "Yes" or "No" response from the user.

an instance of `GtkDialog`. In the simpler example below, we will create a dialog that asks whether the user wants to upgrade the **RGtk2** package installed on the system. Although we could build such a dialog using `GtkDialog` directly, `GtkMessageDialog`, an extension of `GtkDialog`, reduces the amount of necessary code for queries that can be expressed with a textual message and a set of buttons for the response. The dialog is constructed with a single function call:

```
R> main_application_window <- NULL
R> dialog <- gtkMessageDialog(main_application_window, "destroy-with-parent",
+    "question", "yes-no", "Do you want to upgrade RGtk2?")
```

In the above invocation, the first parameter of the call to `gtkMessageDialog` indicates the parent window for the dialog. It is assumed that the main window of the application is stored as `main_application_window`. The second parameter indicates that the dialog should be destroyed when its parent, the main window, is destroyed. The next parameter specifies that this is a *question* dialog, which causes the dialog to display a question mark icon to the left of the text. The predefined set of buttons, in this case consisting of `Yes` and `No`, is given by the next parameter. The final parameter specifies the text of the message. The resulting dialog is shown in Figure 5.

It is desirable for this dialog to be *modal*, meaning that user interaction is restricted to the dialog window until the user responds to the question. By invoking the `gtkDialogRun` function, the dialog becomes modal and execution is blocked until the user gives a response, which is returned from the function. The following code demonstrates the use of `gtkDialogRun`:

```
R> if (dialog$run() == GtkResponseType["yes"]) install.packages("RGtk2")
R> dialog$destroy()
```

If the user answered "Yes", our callback will install the latest version of the **RGtk2** package. The call to `gtkWidgetDestroy` closes the dialog window and renders it unusable, i.e., if the object is used in subsequent computations, an error will be raised, because the dialog widget is no longer valid.

The reference to `GtkResponseType` above is one of the rare cases in which it is necessary to access an enumeration vector to retrieve the numeric value for a nickname. The reason for this is that `gtkDialogGetResponse` returns a plain numeric value to avoid an unnecessary restriction on the number of possible response types from a dialog. This allows programmers to introduce response types that do not exist within the `GtkResponseType` enumeration. In this

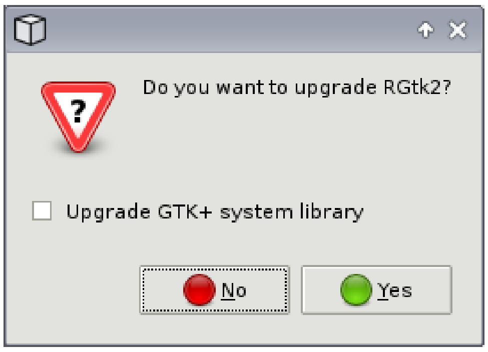


Figure 6: A screenshot of a message dialog with a check box for requesting additional input on top of the original dialog in Figure 5.

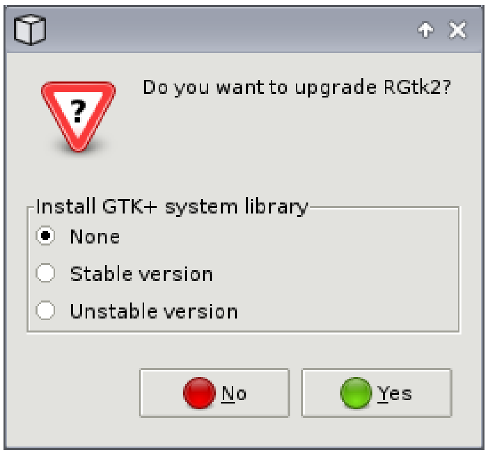
case, it is known from the documentation of `GtkMessageDialog` that the value corresponding to the user clicking the `Yes` button will equal the `yes` value in `GtkResponseType`.

## 3.2. Giving the user more options

Applications often need to ask questions for which a simple "Yes" or "No" answer does not suffice. As the number of possible responses to a query increases, enumerating every response with a button would place a burden on the user with a lengthy sequence of binary questions. It is easy to make a mistake when choosing one response from many and hard to go back to correct such errors. An interface should be forgiving and allow the user to confirm the choice before proceeding. This is how the CRAN mirrors dialog behaves: if the user accidentally chooses a mirror on the other side of the world, the user can correct the choice before clicking the `OK` button and starting the installation process. This relates to the common need for a program to issue a set of queries to the user. Separating each query into its own dialog of buttons may unnecessarily force the user to answer the questions in a fixed, linear order and may not be very forgiving. It would also leave the user without a sense of context. If there were many actions and choices available to the user, a dialog-based interface would be tedious to use, requiring the user to click through dialog after dialog. Instead, a less assertive, non-linear interface is desired. In the examples below, we demonstrate widgets that present options in a passive way, meaning that there is usually no significant, immediate consequence to user interaction with the widget and the user has to conclude the interaction by clicking either the `OK` or `Cancel` button.

The simplest user-level choice is binary and is usually represented in a passive way via a checkbox with a checked/unchecked or on/off state. In **GTK+**, the checkbox class is the `GtkCheckButton`. We may wish to extend our dialog confirming the upgrade of **RGtk2** to include the option of also upgrading the underlying **GTK+** C library. In the snippet below, we achieve this by adding a check button to the dialog. The area above the buttons in the `GtkDialog` is contained within a `GtkVBox`, which is stored as a field named `vbox` in the dialog object. Figure 6 shows our custom checkbox dialog and the following code is how to create it:

```
R> dialog <- gtkMessageDialog(main_application_window, "destroy-with-parent",
```


Do you want to upgrade RGtk2?

Install GTK+ system library
- (●) None
- ( ) Stable version
- ( ) Unstable version

Figure 7: A screenshot of a message dialog with a set of radio buttons on top of the base dialog shown in Figure 5.

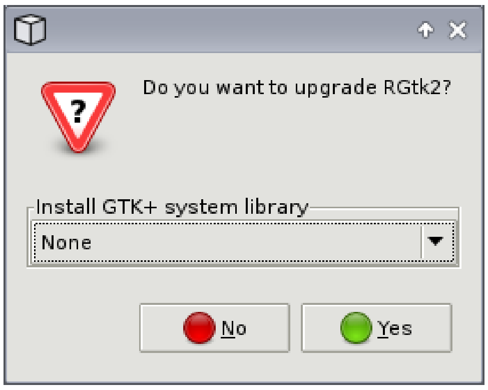
```
+    "question", "yes-no", "Do you want to upgrade RGtk2?")
R> check <- gtkCheckButton("Upgrade GTK+ system library")
R> dialog[["vbox"]]$add(check)
```

Let us now suppose that we would like to give the user the additional option of installing a development (experimental) version of **GTK+**. When an option has several choices, a check button is no longer adequate. A simple approach is to create a set of toggle buttons where only one button may be active at once. The buttons in this set are known as *radio buttons*, corresponding to the pre-programed channel selection buttons on old-style radios. Below, we create a new dialog that asks the user to specify the version of **GTK+** C libraries to install, if any:

```
R> dialog <- gtkMessageDialog(main_application_window, "destroy-with-parent",
+    "question", "yes-no", "Do you want to upgrade RGtk2?")
R> choices <- c("None", "Stable version", "Unstable version")
R> radio_buttons <- NULL
R> vbox <- gtkVBox(FALSE, 0)
R> for (choice in choices) {
+    button <- gtkRadioButton(radio_buttons, choice)
+    vbox$add(button)
+    radio_buttons <- c(radio_buttons, button)
+  }
```

When each radio button is created, it needs to be given the existing collection of buttons already in the group. For creating the first button, NULL should be passed as the group. Each button is added to a vertical box.

A group of radio buttons are often graphically enclosed by a drawn border with a text label indicating the purpose of the buttons. This widget is a container called GtkFrame and is generally used for graphically grouping widgets that are logically related. The code below adds the box containing the radio buttons to a newly created frame:


Do you want to upgrade RGtk2?

Install GTK+ system library

None

Figure 8: A screenshot of a message dialog with a combobox for selecting an option from a drop-down menu before responding to the dialog.

```
R> frame <- gtkFrame("Install GTK+ system library")
R> frame$add(vbox)
R> dialog[["vbox"]]$add(frame)
```

The final result is shown in Figure 7.

Now we would like to go a step further and allow the user to choose the exact version of **GTK+** to install, as **RGtk2** is source compatible with any version from 2.8.0 onwards. As the number of options increases, however, radio buttons tend to consume too much space on the screen. In this case, a label displaying the current selection with a drop down menu allowing for selecting from a list of alternatives may be appropriate. This is known as a `GtkComboBox` in **GTK+**. The following snippet illustrates its use. Each call to `gtkComboBoxAppendText` adds a text item to the drop-down menu. The call to `gtkComboBoxSetActive` makes the first item (0 due to zero-based counting in C) the currently selected one. Figure 8 shows the result and below is the corresponding code:

```
R> dialog <- gtkMessageDialog(main_application_window, "destroy-with-parent",
+    "question", "yes-no", "Do you want to upgrade RGtk2?")
R> choices <- c("None", "GTK+ 2.8.x", "GTK+ 2.10.x", "GTK+ 2.12.x")
R> combo <- gtkComboBoxNewText()
R> combo$show()
R> for (choice in choices) combo$appendText(choice)
R> combo$setActive(0)
R> frame <- gtkFrame("Install GTK+ system library")
R> frame$add(combo)
R> dialog[["vbox"]]$add(frame)
```

## 3.3. The CRAN mirrors dialog

Having demonstrated the creation some basic dialogs, we are now prepared to construct the CRAN mirror selection dialog, shown in Figure 1. Given the large number of CRAN mirrors, one strategy would be to borrow the combobox dialog created above; however, there may be a better alternative. Since there is no reasonable default CRAN mirror, the user always needs to pick a mirror. Packing the mirrors into a combo box would only force the user to make an extra click. Instead, we want to display a reasonable number of CRAN mirrors immediately after the dialog is opened. It may not be possible to display every mirror at once on the screen, but, as seen in the screenshot, we can embed the list in a scrolled box, so that only one part of the list is visible at a given time.

We begin with the construction of the dialog window, as below:

```
R> dialog <- gtkMessageDialog(NULL, 0, "question", "ok-cancel",
+    "Choose a mirror:", show = FALSE)
```

For this dialog, we assume that there is no main application window (see Section 4) to serve as the parent. Instead, we pass `NULL` for the parent and `0` for the second argument rather than `"destroy-with-parent"`. We use the literal `0` here instead of a value name, because `GtkDialogFlags`, like all flag enumerations in **GTK+**, lacks a value for 0.

Next, we create a list for holding the mirror names using the `GtkTreeView` widget (so named because the rows in the list may be organized hierarchically, but we will not discuss this feature). The **RGtk2** package provides a facility for creating a tabular data structure based on an R `data.frame`, called `RGtkDataFrame`. `RGtkDataFrame` is an extension of `GtkTreeModel`, which is the data structure viewed by `GtkTreeView`. Below, we create an `RGtkDataFrame` for our list of CRAN mirrors and construct a `GtkTreeView` based on it:

```
R> mirrors <- read.csv(file.path(R.home("doc"), "CRAN_mirrors.csv"),
+    as.is = TRUE)
R> model <- rGtkDataFrame(mirrors)
R> view <- gtkTreeView(model)
R> view$getSelection()$setMode("browse")
```

The final line configures the `GtkTreeView` so that exactly one item is always selected in the view. This prevents the user from providing invalid input (i.e., a multiple or empty selection).

Initially, the tree view does not contain any columns. We need to create a `GtkTreeViewColumn` to list the mirror names, and we do so with the following code:

```
R> column <- gtkTreeViewColumn("Mirror", gtkCellRendererText(), text = 0)
R> view$appendColumn(column)
```

The first parameter to `gtkTreeViewColumn` specifies the title of the new column. Since we are displaying text (the names of the mirrors) in the column, we pass an instance of `GtkCellRendererText` which draws the values in our `data.frame` as text in the `GtkTreeView`. The parameter named `text` specifies that the first column of the `data.frame` contains the values to draw as text. Note that the index is zero-based (for historical reasons).

Given the large number of CRAN mirrors, the list would take up excessive space if not embedded into a scrolled window. `GtkScrolledWindow` is a container widget that provides a scrolled view of its child when the child requests more space than is available. In the following code, we add the tree view to a `GtkScrolledWindow` instance that requests a minimum vertical size sufficient for showing several mirrors at once:

```
R> scrolled_window <- gtkScrolledWindow()
R> scrolled_window$setSizeRequest(-1, 150)
R> scrolled_window$add(view)
```

The size of `scrolled_window` is set using `gtkWidgetSetSizeRequest`, which takes values in pixel units.

It only remains to add the scrolled window to the dialog, run the dialog, and set the selected CRAN mirror if the user confirms the selection. This is achieved by the following code:

```
R> dialog[["vbox"]]$add(scrolled_window)
R> if (dialog$run() == GtkResponseType["ok"]) {
+    selection <- view$getSelection()
+    sel_paths <- selection$getSelectedRows()$retval
+    sel_row <- sel_paths[[1]]$getIndices()[[1]]
+    options(repos = mirrors[sel_row, "URL"])
+  }
R> dialog$destroy()
```

The selection of a tree view is stored in a separate `GtkTreeSelection` object retrieved by `gtkTreeViewGetSelection`. The `getSelectedRows` method returns a list containing the tree paths for the selected rows and the tree model. The list of tree paths is stored under the name `retval` as it is the actual return value from the C function. Finally, we retrieve the row index from the `GtkTreePath` for the first (and only) selected row and set its URL as the repository.

## 3.4. Embedded R graphics

In a statistical graphical interface, it is often beneficial or necessary to display statistical graphics within the interface. As an example, we consider the contemporary problem of visualizing micoarray data. The large number of genes leads to a significant amount of overplotting when, for example, plotting the expression levels from two chips in a scatterplot. One solution to the problem of overplotting is alpha blending. However, choosing the ideal alpha level may be time-consuming and tedious. Linking a slider widget to the alpha level of an R scatterplot may accelerate the search (See Figures 9 and 10).

As a preliminary step, we use a 2D mixture distribution of correlated variables to simulate expression values for two microarray chips. The following code generates the data:

```
R> n <- 5000
R> backbone <- rnorm(n)
R> ma_data <- cbind(backbone + c(rnorm(3 * (n / 4), sd = 0.1), rt(n/4, 80)),
+    backbone + c(rnorm(3 * (n / 4), , 0.1), rt(n / 4, 80)))
R> ma_data <- apply(ma_data, 2, function(col) col - min(col))
```

The first step towards making our GUI is to create the window that will contain the graphics device and slider widgets:

```
R> win <- gtkWindow(show = FALSE)
```

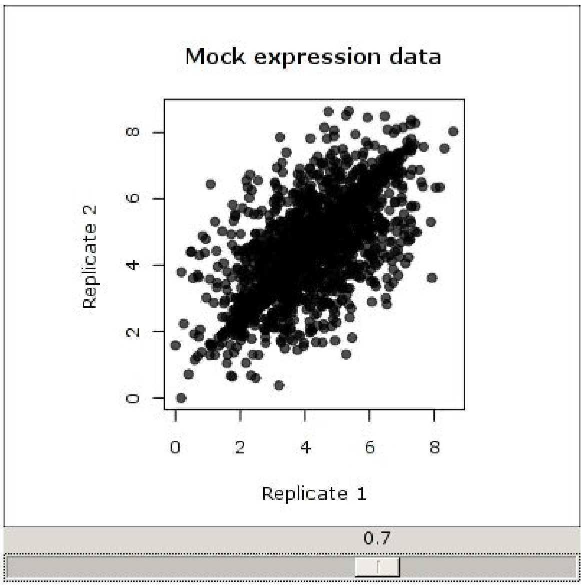


Figure 9: Scatterplot of two microarray replicates, with a slider widget underneath that controls the alpha level of the points. This screenshot shows the initial alpha of 0.7. This value does not lead to a clear display of the density at each location.

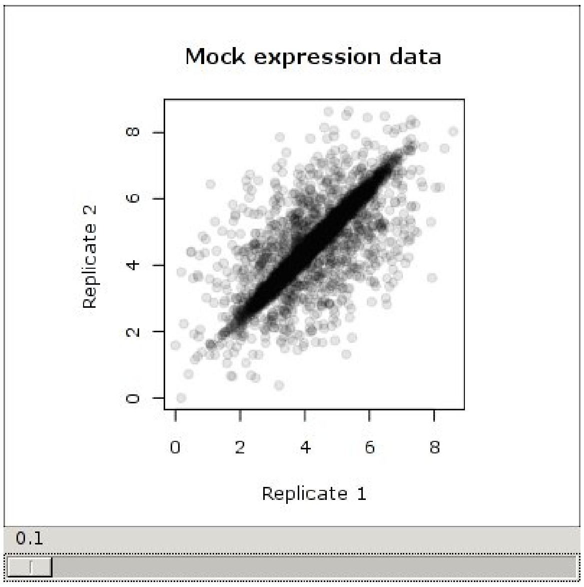


Figure 10: The same scatterplot from 9, except the alpha parameter has been set to to 0.1.

One may embed R graphics within an **RGtk2** GUI using the **cairoDevice** (Lawrence 2009) package. The **cairoDevice** package draws R graphics using **Cairo** (The Cairo Project 2010), a library for vector-based, antialiased graphics. When **cairoDevice** draws to the screen it is actually drawing to a **GTK+** widget of type `GtkDrawingArea`. A `GtkDrawingArea` is an empty widget meant for drawing arbitrary graphics in an interface. Here we construct a drawing area in which the R graphics will be drawn:

```
R> graphics <- gtkDrawingArea()
```

Now that we have a widget for displaying R graphics, we need the slider that controls the alpha level. A slider is a widget, much like a scroll bar, for choosing a number at a certain precision from a certain range. Here, a horizontal slider, called `GtkHScale`, is created with a range from 0.1 to 1.0, with a step size of 0.1:

```
R> slider <- gtkHScale(min = 0.1, max = 1.00, step = 0.1)
```

When the user moves the slider, the plot should be updated so that its alpha level reflects the slider value. This is achieved by connecting an R callback function to the `value-changed` signal of the slider, as in the code below:

```
R> scale_cb <- function(range) {
+    par(pty = "s")
+    plot(ma_data[, 1], ma_data[, 2],
+    col = rgb(0, 0, 0, alpha = range$getValue()),
+    xlab = "Replicate 1", ylab = "Replicate 2",
+    main = "Mock expression data", pch = 19)
+  }
R> gSignalConnect(slider, "value-changed", scale_cb)
```

The callback function, `scale_cb`, replots the microarray data, `ma_data`, using an alpha level equal to the current value of the slider.

The next steps are to add the drawing area and the slider to the window and then to show the window on the screen. Although the window is a container, it inherits from `GtkBin`, meaning that it can hold only a single child widget. Thus, we will pack our widgets into a vertical stacking box container, `GtkVBox`, and add our box to the window. Here, we would like the graphics to take up all of the space not consumed by the slider, so the graphics device is packed to `expand` and `fill`, while the slider is not (see Section 2.3). The following code performs the packing operation:

```
R> vbox <- gtkVBox()
R> vbox$packStart(graphics, expand = TRUE, fill = TRUE, padding = 0)
R> vbox$packStart(slider, expand = FALSE, fill = FALSE, padding = 0)
R> win$add(vbox)
```

As a final step, we set the default size of the window and show it and all of its children:

```
R> win$setDefaultSize(400,400)
R> win$showAll()
```

Now that the window is visible on screen, we can instruct R to draw its graphics to the drawing area using the `asCairoDevice` function in the **cairoDevice** package:

```
R> require("cairoDevice")
R> asCairoDevice(graphics)
```

The call to `asCairoDevice` creates an R graphics device from our drawing area widget and makes the device active, so that it is the target of R plotting commands.

Finally, the value of the slider is initialized to 0.7,

```
R> slider$setValue(0.7)
```

which in turn activates the callback, generating the initial plot. The initial state of the interface is shown in Figure 9. Figure 10 shows the plot after the user has moved the slider to set the value of alpha to 0.1.

## 4. Sample application

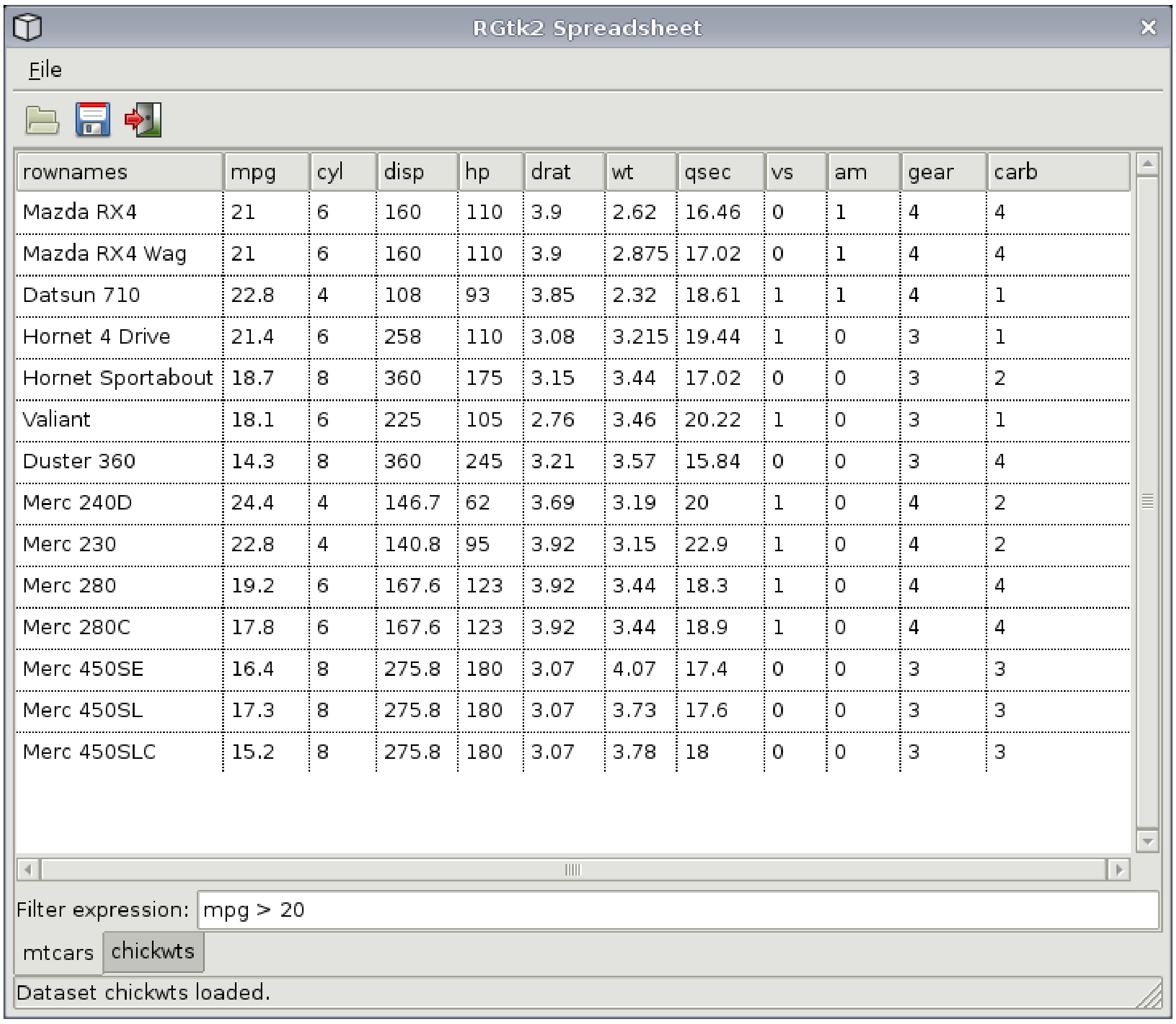
The interfaces presented thus far are each designed for a singular, focused task, such as choosing a CRAN mirror or viewing a scatterplot at different alpha levels. However, an interface often supports a larger collection of separate operations, and the user is in control of initiating different tasks from the general interface. These interfaces for broader, more complex applications are typically based on what is called an *application window*, which often contains a menu bar, tool bar, application-specific area, and status bar in order from top to bottom. The menu bar and tool bar are widgets designed to facilitate the user selecting different *actions*, each of which represents an option or operation in the application. The status bar at the bottom commonly reports information about the activities or state of the application or information for the user as a text message and may be adjacent to a progress bar that displays the continuing progress of long running operations. This layout and design is a common convention which helps users navigate a new GUI.

The following example demonstrates how one might construct a reasonably complex application using **RGtk2**. We aim to build a viewer for one or more R `data.frame`s that is capable of sorting and filtering the rows in each `data.frame`. We also give it facilities to load and save a `data.frame` to and from a CSV file.

The resulting GUI is shown in Figure 11. Each `data.frame` frame is displayed in a table, using a `GtkTreeView` widget. As we would like to support multiple spreadsheets at once, we embed each table in a tabbed notebook, `GtkNotebook`. Below each spreadsheet is a text entry (a `GtkEntry` widget), in which the user may enter an expression for filtering the table view. Below this is a status bar (a `GtkStatusbar` widget) that communicates the status of the application to the user, such as whether the loading of a dataset is complete. At the top are a menu bar (a `GtkMenubar` widget) and tool bar (a `GtkToolbar` widget) that allow the user to invoke various actions, such as loading a new dataset or quitting the application.

### 4.1. Main window

We begin by creating the main window for the application and setting its default size, specified in number of screen pixels:



| rownames | mpg | cyl | disp | hp | drat | wt | qsec | vs | am | gear | carb |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mazda RX4 | 21 | 6 | 160 | 110 | 3.9 | 2.62 | 16.46 | 0 | 1 | 4 | 4 |
| Mazda RX4 Wag | 21 | 6 | 160 | 110 | 3.9 | 2.875 | 17.02 | 0 | 1 | 4 | 4 |
| Datsun 710 | 22.8 | 4 | 108 | 93 | 3.85 | 2.32 | 18.61 | 1 | 1 | 4 | 1 |
| Hornet 4 Drive | 21.4 | 6 | 258 | 110 | 3.08 | 3.215 | 19.44 | 1 | 0 | 3 | 1 |
| Hornet Sportabout | 18.7 | 8 | 360 | 175 | 3.15 | 3.44 | 17.02 | 0 | 0 | 3 | 2 |
| Valiant | 18.1 | 6 | 225 | 105 | 2.76 | 3.46 | 20.22 | 1 | 0 | 3 | 1 |
| Duster 360 | 14.3 | 8 | 360 | 245 | 3.21 | 3.57 | 15.84 | 0 | 0 | 3 | 4 |
| Merc 240D | 24.4 | 4 | 146.7 | 62 | 3.69 | 3.19 | 20 | 1 | 0 | 4 | 2 |
| Merc 230 | 22.8 | 4 | 140.8 | 95 | 3.92 | 3.15 | 22.9 | 1 | 0 | 4 | 2 |
| Merc 280 | 19.2 | 6 | 167.6 | 123 | 3.92 | 3.44 | 18.3 | 1 | 0 | 4 | 4 |
| Merc 280C | 17.8 | 6 | 167.6 | 123 | 3.92 | 3.44 | 18.9 | 1 | 0 | 4 | 4 |
| Merc 450SE | 16.4 | 8 | 275.8 | 180 | 3.07 | 4.07 | 17.4 | 0 | 0 | 3 | 3 |
| Merc 450SL | 17.3 | 8 | 275.8 | 180 | 3.07 | 3.73 | 17.6 | 0 | 0 | 3 | 3 |
| Merc 450SLC | 15.2 | 8 | 275.8 | 180 | 3.07 | 3.78 | 18 | 0 | 0 | 3 | 3 |

Filter expression: mpg > 20

Dataset chickwts loaded.

Figure 11: Screenshot of a spreadsheet application constructed with **RGtk2**. The current sheet is from the `mtcars` dataset. The table is filtered by the expression `mpg > 20` and sorted by in decreasing order of the values of the `mpg` variable.

```
R> main_window <- gtkWindow(show = FALSE)
R> main_window["title"] <- "RGtk2 Spreadsheet"
R> main_window$setDefaultSize(600, 600)
```

## 4.2. Menu bar and tool bar

Our spreadsheet application will support three user actions: open, save and quit. All three actions will be made available in both the menu bar and tool bar. Providing a user action requires (1) implementing a callback function, (2) defining the action properties, and (3) manifesting the action as a widget in the GUI. We consider each of these steps in turn.

*Implementing the callbacks*

We begin by implementing the callbacks corresponding to the user actions. operations for the menu items and corresponding callbacks to load and save a `data.frame` and to quit the "application":

```
R> open_cb <- function(widget, window) {
+    dialog <- gtkFileChooserDialog("Choose a CSV file", window, "open",
+      "gtk-cancel", GtkResponseType["cancel"], "gtk-open",
+      GtkResponseType["accept"])
+    if (dialog$run() == GtkResponseType["accept"]) {
+      df <- read.csv(dialog$getFilename())
+      load_spreadsheet(df, basename(dialog$getFilename()))
+    }
+    dialog$destroy()
+  }
R> save_cb <- function(widget, window) {
+    dialog <- gtkFileChooserDialog("Enter a name for the file", window,
+      "save", "gtk-cancel", GtkResponseType["cancel"], "gtk-save",
+      GtkResponseType["accept"])
+    if (dialog$run() == GtkResponseType["accept"])
+      save_file(dialog$getFilename())
+    dialog$destroy()
+  }
R> quit_cb <- function(widget, window) window$destroy()
```

Each of these functions is a callback which takes the widget associated with the action as its first argument and the top-level window as its second. The load and save operations leverage the `GtkFileChooserDialog` widget type, a dialog that contains a graphical file browser for specifying the path to a file. `GtkFileChooserDialog` has several modes corresponding to common file selection tasks. In this case, we use the `open` mode for the reading action and the `save` mode for the writing action. The `accept` response from the dialog indicates that the user has confirmed the file selection by clicking the `Open` or `Save` button.

*Defining the actions*

We now define the actions that will delegate to the callbacks implemented above. The `GtkAction` class represents an operation that a user may request an application to perform. A `GtkAction` instance may be manifested as a widget in multiple ways, such as an item in a menu or a button in a tool bar. The widgets are synchronized with the properties of the `GtkAction`. For example, if an action is disabled, the menu items and tool bar buttons will also be disabled. Extensions of `GtkAction` exist for toggle and radio options, but those are not described here. For details, see `help(GtkToggleAction)` and `help(GtkRadioAction)`, respectively. A `GtkActionGroup` is a container for `GtkAction` objects.

In the following code, we define the actions for our application and bundle them into a `GtkActionGroup`:

```
R> actions <- list(
+    list("FileMenu", NULL, "_File"),
+    list("Open", "gtk-open", "_Open File", "<control>O",
+      "Select a CSV file to load as a spreadsheet", open_cb),
+    list("Save", "gtk-save", "_Save", "<control>S",
+      "Save the current spreadsheet to a CSV file", save_cb),
+    list("Quit", "gtk-quit", "_Quit", "<control>Q",
+        "Quit the application", quit_cb)
+  )
R> action_group <- gtkActionGroup("spreadsheetActions")
R> action_group$addActions(actions, main_window)
```

Above, each action is defined with a list, containing the action ID (for referring to the action later in the code), icon ID, label, keyboard shortcut, tooltip, and callback. The first action will serve as the basic menu container for the rest of the items and actions. Since it performs no function, it is not necessary to specify all of the fields, such as the callback.

Specifying the action definitions as R lists is an example of high-level type conversion, where a native R structure is implicitly converted to a complex **GTK+** object. In this case, an R list is being converted to a set of `GtkAction` objects. See Section 6.3 for a technical explanation and justification.

*Creating widgets for the actions*

In order to make these actions available to the user, they need to be mapped to a widget in the GUI. **GTK+** provides a class known as `GtkUIManager` to facilitate this operation. In the following code, we create a `GtkUIManager` instance and register our actions with it:

```
R> ui_manager <- gtkUIManager()
R> ui_manager$insertActionGroup(action_group, 0)
```

Next, we specify the layout of the menu bar and tool bar containing the actions defined above, by calling the `gtkUIManagerAddUi` method:

```
R> merge <- ui_manager$newMergeId()
R> ui_manager$addUi(merge.id = merge, path = "/", name = "menubar",
+    action = NULL, type = "menubar", top = FALSE)
R> ui_manager$addUi(merge, "/menubar", "file", "FileMenu", "menu", FALSE)
R> ui_manager$addUi(merge, "/menubar/file", "open", "Open", "menuitem", FALSE)
R> ui_manager$addUi(merge, "/menubar/file", "save", "Save", "menuitem", FALSE)
R> ui_manager$addUi(merge, "/menubar/file", "sep", NULL, "menuitem", FALSE)
R> ui_manager$addUi(merge, "/menubar/file", "quit", "Quit", "menuitem", FALSE)
R> ui_manager$addUi(merge, "/", "toolbar", NULL, "toolbar", FALSE)
R> ui_manager$addUi(merge, "/toolbar", "open", "Open", "toolitem", FALSE)
R> ui_manager$addUi(merge, "/toolbar", "save", "Save", "toolitem", FALSE)
R> ui_manager$addUi(merge, "/toolbar", "quit", "Quit", "toolitem", FALSE)
```

Each piece of the user interface added to a `GtkUIManager` instance must be associated with a *merge id*, as retrieved from `gtkUIManagerNewMergeId`. It is passed to `gtkUIManagerAddUi` as the `merge_id` parameter. This allows removing (unmerging) the UI in batch at a later time. The `path` parameter indicates where the UI element should be merged. Similar to a path in a file system or URL, each element name in the path is delimited by a forward slash ("/"). The name parameter identifies the element to the manager, and action is the ID of the action in the provided action group. The ID can be NULL in the case of containers (menu bars and tool bars) and separators. Finally, type indicates the type of UI element, such as a tool bar or menu bar. The default is auto, which asks the GtkUIManager to guess based on the path.

The next step is to use the GtkUIManager to create the actual menu bar and tool bar widgets from the action definitions and layout.

```
R> menubar <- ui_manager$getWidget("/menubar")
R> toolbar <- ui_manager$getWidget("/toolbar")
R> main_window$addAccelGroup(ui_manager$getAccelGroup())
```

The final line above enables keyboard shortcuts in the main window.

### 4.3. Status bar

To report information from and about the application, we will use a GtkStatusbar widget. A status bar maintains a stack of text messages and displays the message on top of the stack. When a message is added to the status bar stack, it is immediately displayed, and when the message is removed, the previous one is displayed. Each message is associated with a *context*, and each context has its own ID. A context ID is a number that is generated in a consistent way from any user-supplied string that serves as the human-readable context name. A context ID may be created using the gtkStatusbarGetContextId function. Here we create a status bar and push the message "Ready" onto the top of the stack within the context named "info" (other contexts could be named, e.g., "warning" or "error"):

```
R> statusbar <- gtkStatusbar()
R> info <- statusbar$getContextId("info")
R> statusbar$push(info, "Ready")
```

### 4.4. Spreadsheet panel

Next, we need to create the GtkTreeView that will display a given data.frame as a table. The data is first loaded into a GtkTreeModel, from which the GtkTreeView retrieves the values it displays. Below each sheet is a text entry box for entering an expression for filtering the spreadsheet rows. All of the open sheets will be placed within notebook-style container.

#### *Data model*

The following function, create_tree_model, will create a GtkTreeModel object that obtains its data from an R data.frame, passed as an argument to the function:

```
R> create_tree_model <- function(df) {
+    df <- cbind(rownames = rownames(df), df)
+    filter_df <- cbind(filter = TRUE, df)
+    model <- rGtkDataFrame(filter_df)
+    filter_model <- gtkTreeModelFilterNew(model)
+    filter_model$setVisibleColumn(0)
+    sort_model <- gtkTreeModelSort(filter_model)
+    sort_model
+  }
```

The function employs the `RGtkDataFrame` utility that allows the `GtkTreeView` to use an R `data.frame` as its data source. term. In order to support filtering and sorting of the displayed data, the `RGtkDataFrame` is proxied by a `GtkTreeModelFilter` model, which in turn is proxied by a `GtkTreeModelSort` model. A proxy data model sits between a source data model and a client, such as a `GtkTreeView`. The data provided by a proxy model results from the modification of the data in the source model.

### *Table view*

The next function, `create_tree_view`, will create the `GtkTreeView` given the `GtkTreeModel` created by `create_tree_model` above:

```
R> create_tree_view <- function(model) {
+    tree_view <- gtkTreeView(model)
+    rdf <- model$getModel()$getModel()
+    sapply(tail(seq_len(ncol(rdf)), -1), function(j) {
+      renderer <- gtkCellRendererText()
+      column <- gtkTreeViewColumn(colnames(rdf)[j], renderer, text = j - 1)
+      column$setSortColumnId(j - 1)
+      column$setCellDataFunc(renderer,
+        function(column, renderer, model, iter)
+      {
+        iter <- model$convertIterToChildIter(iter)$child.iter
+        child <- model$getModel()
+        iter <- child$convertIterToChildIter(iter)$child.iter
+        i <- rdf$getPath(iter)$getIndices()[[1]] + 1
+        renderer["text"] <- as.character(rdf[i, j])
+      })
+      tree_view$appendColumn(column)
+    })
+    tree_view$setHeadersClickable(TRUE)
+    if (is.null(gtkCheckVersion(2, 10, 0))) tree_view$setGridLines("both")
+    tree_view
+  }
```

Above, each column of the `data.frame`, provided as the second argument, is displayed by a column in the tree view. We configure the tree view so that it shows grid lines (if the user has **GTK+** 2.10.0 or higher) and supports sorting on a column when the user clicks on the column header. The call to `setCellDataFunc` attaches a callback that formats the text values as R does by default (**GTK+** takes a simpler approach that gives each number 6 significant figures). Note that this callback is called each time a cell is rendered, so it could negatively impact performance, especially when scrolling. For large spreadsheets, we recommend using a dedicated spreadsheet application.

*Filter text entry*

Next, we define a function that creates the text box for the user to enter a filter expression:

```
R> create_entry <- function(model) {
+    entry <- gtkEntry()
+    gSignalConnect(entry, "activate", function(entry) {
+      model[, "filter"] <<- eval(parse(text = entry$text),
+        as.data.frame(model))
+      })
+    entry
+  }
```

This uses the `GtkEntry` widget. Whenever the `GtkEntry` is "activated," e.g., by the user pressing the `ENTER` key, we update the filter by the result of the R expression.

*Notebook of sheets*

In order to handle multiple spreadsheets simultaneously but display only one at a time, we will use a special type of container called `GtkNotebook`. This provides tabs on the border of the notebook like a ring binder which the user can select to switch between the different widgets within the notebook. This is used in Excel to present several work sheets within a single window, and also within certain Web browsers to allow the user to view multiple Web pages without opening multiple windows.

Below, we create the notebook:

```
R> notebook <- gtkNotebook()
R> notebook$setTabPos("bottom")
```

## 4.5. Integrating the components

The menu bar, tool bar, spreadsheet notebook and status bar all need to be contained within the main window. In the code below, we add each of these components to the GUI by placing them inside a `GtkVBox`:

```
R> vbox <- gtkVBox(homogeneous = FALSE, spacing = 0)
R> vbox$packStart(menubar, expand = FALSE, fill = FALSE, padding = 0)
R> vbox$packStart(toolbar, FALSE, FALSE, 0)
R> vbox$packStart(notebook, TRUE, TRUE, 0)
R> vbox$packStart(statusbar, FALSE, FALSE, 0)
R> main_window$add(vbox)
R> main_window$show()
```

## 4.6. Loading a spreadsheet

Finally, we define the `load_spreadsheet` function, as called from the `open_cb` callback above, that loads a `data.frame` into the GUI:

```
R> load_spreadsheet <- function(df, name) {
+    model <- create_tree_model(df)
+    tree_view <- create_tree_view(model)
+    entry <- create_entry(model$getModel()$getModel())
+
+    hbox <- gtkHBox(FALSE, 5)
+    hbox$packStart(gtkLabel("Filter expression:"), FALSE, FALSE, 0)
+    hbox$packStart(entry, TRUE, TRUE, 0)
+    vbox <- gtkVBox(FALSE, 5)
+    scrolled_window <- gtkScrolledWindow()
+    scrolled_window$add(tree_view)
+    vbox$packStart(scrolled_window, TRUE, TRUE, 0)
+    vbox$packStart(hbox, FALSE, FALSE, 0)
+
+    if (missing(name)) name <- deparse(substitute(df))
+    notebook$appendPage(vbox, gtkLabel(name))
+
+    statusbar$push(info, paste("Dataset", name, "loaded."))
+  }
```

This function creates the necessary widgets and packs them into a notebook page. To limit its visible size, the data grid/table is added to a `GtkScrolledWindow`. The function concludes by updating the status bar to indicate that the dataset has been successfully loaded.

An example of using the above function to add a spreadsheet is given below:

```
R> load_spreadsheet(mtcars)
```

This application is obviously missing many important features. For example, there is no easy way to return to the complete `data.frame` after subsetting, and it is not possible to edit the cells. The main purpose of the example is to introduce the process of building an application window.

## 5. Advanced features

This section describes features of **RGtk2** that are beyond the construction of basic and intermediate GUIs. It is meant for readers interested in advanced and specialized **RGtk2** features such as the ability to extend **GTK+** classes and interface with low-level and third-party libraries that are integrated with **GTK+**. Much of this functionality is applicable outside of GUI construction.

First, we describe the additional libraries (other than **GTK+**) bound by **RGtk2** that are meant to support the construction of advanced, graphically-intensive interfaces. The focus then shifts to the low-level support for the **GObject** object-oriented programming library. The **RGtk2** user is able to manipulate objects in external **GObject**-based applications (i.e., top-level GUIs running within the same R session) that are bound to R by code outside of the **RGtk2** package. **RGtk2** also supports defining new **GObject** classes in R.

### 5.1. Additional library support

The **GTK+** 2.0 library incorporates several other libraries: **Cairo**, **GDK**, **GdkPixbuf**, **Pango** and **ATK**. The **RGtk2** package provides R-level bindings for these libraries, in addition to **GTK+** itself. The following sections describe the purpose and functionality of each library.

#### *Cairo*

Cairo is a 2D vector graphics library with which **GTK+** widgets are drawn. It is possible to use **Cairo** directly to draw custom graphics within a `GtkDrawingArea`. The library is also useful outside of GUI construction, in that one can draw vector graphics to off-screen surfaces in common formats such as PNG, SVG, PS, and PDF files.

#### *GDK*

The GIMP Drawing Kit, **GDK**, is the low-level hardware access and drawing layer for **GTK+**. It is most useful for raster-based (non-vector) drawing of graphical primitives like lines, rectangles and circles and for handling raw mouse and keyboard events. It also provides access to windowing system resources, such as screens in a multi-headed environment. Although the drawing functions of **GDK** overlap somewhat with **Cairo**, **Cairo** is for drawing vectors, while **GDK** is for direct drawing of pixels. Another reason for the redundancy is that **GDK** predates **Cairo**, and thus the **GDK** drawing routines are present for backwards compatibility.

#### *GdkPixbuf*

**GdkPixbuf** is an image manipulation library based on **GDK**. Its features include rendering, scaling, and compositing of images. **GdkPixbuf** can read and write several image formats, including JPEG, PNG, and GIF. Like **Cairo**, **GdkPixbuf** could be used independently of a GUI for working with arbitrary graphics in R.

#### *Pango*

**Pango** provides facilities for rendering and formatting text with rich capabilities for handling international characters. It also provides cross-platform access to the font configuration of a system. **Pango** is most often used directly for embedding text in graphics when drawing to a `GtkDrawingArea` or an off-screen destination, e.g., image.

#### *ATK*

The Accessibility ToolKit (**ATK**) supports accessibility technologies to make GUIs amenable to users with "disabilities". It allows accessibility devices to interact with **GTK+** GUIs. **ATK** is not likely to be very useful from R. Its binding is included for the sake of completion, since **ATK** types are present in the **GTK+** API.

#### *Libglade*

**Libglade** constructs **GTK+** GUIs from XML descriptions. The XML descriptions are output from **Glade**, which is a GUI tool for interactively designing other GUIs. As of **GTK+** 2.12.0, which includes native support for constructing widgets from XML descriptions, **Libglade** is essentially obsolete. The bindings are still included for backwards compatibility.

## 5.2. GObject primer

**GTK+**, as well as the libraries described in the previous section, except for **Cairo**, are based on the **GObject** library for object-oriented programming in C. **GObject** forms the basis of many other open-source projects, including the GNOME (Warkus 2004) and **XFCE** (Fourdan 2000) desktops and the **GStreamer** multimedia framework (Walthinsen 2001).

**RGtk2** interfaces with parts of **GObject** and permits the R programmer to create new **GObject** classes in R. Understanding this functionality depends on a familiarity with the concepts underlying **GObject**. This section introduces those concepts.

**GObject** is organized as a collection of modules. The fundamental modules are `GType`, `GSignal`, and the base `GObject` class. Each of these modules is described in the following sections. For further details, please see the **GObject** documentation (Breuer *et al.* 2010).

### `GType`

`GType` is at the core of **GObject**. Its basic purpose is to manage the definition, registration and introspection of types at run-time. The main commonality between all `GType`s is that they define a method for copying their values. This allows generic memory management for every value with a `GType`. Those `GType`s that directly define a copy mechanism, instead of inheriting one, are known as *fundamental* `GType`s.

The set of fundamental `GType`s includes many of the built-in C data types. For example, "primitive" types like integers, doubles, and strings (character pointers) are all fundamental `GType`s.

Arbitrary C structures are adapted to the `GType` framework by providing a copy function and free function for the structure. Such `GType`s are said to be *boxed* and inherit from the fundamental `GType` called `GBoxed`. For example, **RGtk2** registers a boxed `GType` for the R `SEXP` structure, which is used to represent all R objects in C.

The `GType` module also supports the definition of object types (the main purpose of **GObject**). Like all `GType`s, object `GType`s must be or inherit from a fundamental `GType`. **GObject** provides a fundamental `GType`, `GObject`, that may be extended to define a new type of object. Every `GType` derived from `GObject` has a C structure representing its class. Inheritance of class structures is accomplished through the standard C idiom for object-oriented programming: prefixing a structure with the structure of the parent class, so that fields are aligned. The use of the structure prefixing idiom restricts **GObject** to single inheritance. The class structure contains class-wide fields, including function pointers called *virtual functions* that may be overriden by changing the value of the corresponding field in the class structure during intialization. This is the primary mechanism in **GObject** for changing class behavior through inheritance. Each object `GType` also has a registered structure with instance-level members (i.e., fields). The instance structure of a `GType` inherits from the parent instance structure using the same idiom as the class structures. An instance of an object `GType` is manifested as a value of the corresponding instance structure. In order to link an instance to its class, each instance structure holds a reference to the shared value of the class structure for the `GType`.

Like many object-oriented languages, **GObject** supports the definition and implementation of *interfaces*. An interface specifies a set of methods that represent a role performed by one or more classes, where the role is shared independently of the class hierarchy. If a class plays a role represented by an interface, it may formally declare the contract by registering itself as an implementation of the interface. As a result, the type is required to provide values (implementations) for the methods declared by the interface. Any object `GType`, such as `GObject`, may implement multiple interfaces. Like a `GObject`-derived `GType`, an interface has a class structure that declares its virtual functions (i.e., methods). Every interface class structure may only be prefixed by `GTypeInterface`, so there is no inheritance between interfaces in **GObject**. This is a significant difference from many object-oriented languages. However, an interface can be made to *require* the implementation of one or more other interfaces by any `GType` that implements it. Unlike `GObject`, `GTypeInterface` is non-instanciable, so there is no instance-level structure and it is not possible to create instances of interfaces directly.

Two other fundamental `GTypes` are `GEnum` and `GFlags`, both of which are registered with a class structure. The `GEnumClass` structure stores metadata about a particular enumeration, such as the names and nicknames of its values. `GFlags` is similar as it represents an enumeration where the values are intended to be combined bitwise (via `AND` and `OR` operations) to represent the presence of one or more settings.

#### `GSignal`

One of the defining characteristics of **GObject** is its emphasis on *signals*, which were introduced earlier in this paper in the context of notification of user events in a **GTK+** GUI. Any instance of a `GType` can have registered signals. Each *signal* is defined by its name and the types of its arguments and return value. A class inherits signals from its parents.

#### `GObject` *base class*

`GObject` is the basic/fundamental classed and instanciable `GType` provided by the **GObject** library. The key feature provided by the `GObject` class, from the perspective of the **RGtk2** user, are *properties*. Properties may be thought of as introspectable and encapsulated public fields. Like instance fields of a `GObject`-derived `GType`, properties are inherited. They support automated validation of their values at runtime, and a change in a property value emits the `notify` signal from its instance, allowing objects to respond to changes in the state of other objects. It is possible to control whether a property is readable, writeable, and more. Depending on the options specified in the declaration of a property, one may be able to or even restricted to set a property at construction time, using the generic `GObject` constructor, `gObject()`.

A property is defined by a `GParamSpec` structure that specifies a name, nickname, description, value `GType`, and other options. There are subclasses of `GParamSpec` for particular `GTypes` that permit specification of further constraints. For example, `GParamSpecInt` is specific to integers and can be configured to restrict its valid range of integer values between a minimum and maximum. Many `GParamSpec` subclasses also permit default values.

## 5.3. Interfacing with external GObject-based applications

Many of the **RGtk2** functions developed for the creation of GUIs using **GTK+** are applicable to other libraries and applications based on **GObject**. There are several such packages of interest to staticians, including **Gnumeric**, a spreadsheet application, and **GGobi**, software for multivariate interactive graphics. The **rggobi** package (Temple Lang and Swayne 2001) provides a high-level interface to **GGobi** from R. Although it is somewhat hidden, **rggobi** objects are `externalptr`s that reference the underlying **GGobi** objects, which extend `GObject`. **RGtk2** uses the same R representation, so many **RGtk2** functions can operate on **rggobi** objects directly without additional interface code.

As an example, we consider the problem of displaying an R plot in response to a user "identifying" a point in a **GGobi** plot with the mouse. When a **GGobi** point is identified, the main **GGobi** context emits the `identify-point` signal. If we connect an R function to this signal, using `gSignalConnect`, the function will be executed whenever a point is identified. The following code displays data within a **GGobi** window and draws a fit of the simple linear model in a separate R graphics window in response to the user identifying a point:

```
R> library("rggobi")
R> attach(mtcars)
R> gg <- ggobi(mtcars)
R> model <- lm(mpg ~ hp)
R> plot(hp, mpg)
R> abline(model)
R> gSignalConnect(gg, "identify-point",
+    function(gg, plot, id, dataset) {
+      plot(hp, mpg)
+      points(hp[id + 1], mpg[id + 1], pch = 19)
+      abline(model)
+    })
```

The **GGobi** instance is initialized with the `mtcars` dataset. A linear model is fit with `lm` and the line is drawn on an R plot. The important step is connecting a handler to the `identify-point` signal. The handler regenerates the R plot, and, for the identified point, replaces the empty circle glyph with a filled circle. In this way, we have created a simple integration of the interactive graphics of **GGobi** with an R graphic that displays a linear model fit, which **GGobi** cannot display. Since the **GGobi** GUI is based on **GTK+**, it would also be possible to embed the **GGobi** plot into an **RGtk2** GUI. More interesting integration uses the same basic tools. Please see the **rggobi** documentation for more details.

## 5.4. Defining GObject classes

All of the above examples utilize objects that are implemented in C. **RGtk2** supports the definition of `GObject`-derived classes from within R. The `gClass` function in R registers a class, given the name of the new class, the name of the parent class, and the class definition. The class definition is a series of arguments that specify the new fields, new methods, methods that override inherited methods, signals, properties, and initialization function for the class. The name of a parameter specifies its role in the definition.

### *Example of defining a class*

The example below illustrates the definition of a new `GObject`-derived class by revisiting the example in Section 3.4 involving the embedded plotting of microarray data. The slider in that example controls the alpha level of the points in the scatterplot in a linear fashion. Given the large amount of overplotting, the alpha level does not have a strong visual effect until it approaches its lower limit. One may desire greater control in this region, without limiting the range of the slider.

A possible solution would be to map the slider value to an alpha value using a non-linear function. All that is required is to change the slider callback so that it computes the alpha value as a non-linear function of the slider value. However, the label on the slider would be inaccurate; it would still report the original value. Overriding how the label is computed is possible by connecting a handler to the `format-value` signal on the `GtkScale` class. Let us assume, however, that we would like to create a reusable type of slider that mapped its value using a specified R expression.

Below is our invocation of `gClass` that defines `RTransformedHScale`, an extension of `GtkHScale`, the horizontal slider:

```
R> tform_scale_type <- gClass("RTransformedHScale", "GtkHScale",
+    .props = list(
+      gParamSpec(type = "R", name = "expr", nick = "e",
+        blurb = "Transformation of scale value",
+        default.value = expression(x))
+    ),
+    .public = list(
+      getExpr = function(self) self["expr"],
+      getTransformedValue = function(self)
+        self$transformValue(self$value)
+    ),
+    .private = list(
+      transformValue = function(self, x) eval(self$expr, list(x = x))
+    ),
+    GtkScale = list(
+      format_value = function(self, x) as.character(self$transformValue(x))
+    )
+  )
```

The third argument to `gClass`, `.props`, is a list containing property definitions. Each property is defined by a `GParamSpec` structure created using the `gParamSpec` function. `RGtkTransformedHScale` defines a single property named `expr` for holding the R `expression` that performs the transformation, e.g., `x^3`. Definitions of properties may refer to any `GType` by name. The names of primitive R types, like `integer` and `character` are mapped to the corresponding (scalar) `GType`, if available. It is also possible to specify the `RGtkSexp` type, as we have done for `RGtkTransformedHScale` using the shorthand alias `R`. The Values of type `RGtkSexp` are left as native R objects instead of being converted to a C type, allowing the storage of R types that do not have a conventional C analog, like expressions, data frames, fitted models and S4 objects. For `RGtkSexp` properties, it is possible to specify the underlying R type for validation purposes. In our example, that type is inferred from the default value, which is of mode `expression`. The `any` type allows an `RGtkSexp` property to hold any R type. Normally, the class definining a property is responsible for handling the getting and setting of it. In order to override the management of a property defined by a parent class, the name of the property should be included in a character vector passed as an argument named `.prop_overrides` to the `gClass` function.

Methods and fields may be encapsulated at the public, protected or private level. Public members may be accessed by any code, while protected members are restricted to methods belonging to the same class or a subclass. Access to private members is the most restricted as they are only available to methods in the same class. `gClass` has a separate parameter for each level of encapsulation. The values should be lists and are named according to their level of encapsulation: `.public`, `.protected` or `.private`. The functions for the methods and the initial assignments for the fields should be passed in the relevant parameter. The name of a member in a list serves as its identifier. In our example above, we define two public methods, `getExpr` and `getTransformedValue`, for retrieving the transformation expression and the transformed value, respectively. There is one private method, `transformValue` that is a utility for evaluating the expression on the current value.

Any virtual function defined by an inherited class or registered interface may be overriden. Like methods, virtual functions are implemented as R functions. In the `RGtkTransformedHScale` example, we override the `format_value` virtual function in the `GtkScale` class to display the transformed value in the label above the slider. We first define the R function that implements the new behavior. Next, since the `gClass` function requires all overrides of methods from a particular class to be grouped together in a list, we create a list for `GtkScale`. We then add our R function to the list as an element named `format_value`. This informs `gClass` that we are overriding the `format_value` method.

Any public or protected method defined in R may be overridden in R as if it were a virtual function. This is useful when the new class extends a class that itself is defined in R. Methods external to R may only be overridden if they are virtual functions.

A function implementing a virtual function may delegate to the function that it overrides from an ancestor class. This is achieved by calling the `parentHandler` function and passing it the name of the method and the arguments to forward to the method. For example, in the override of `format_value` in the `RGtkTransformedHScale` class, we could call `parentHandler("format_value", self, x)` to delegate to the implementation of `format_value` in `GtkScale`.

Two elements of the class definition that are not in the example above are the list of signal definitions and the initialization function. The signal definition list is passed as a parameter named `.signals` and contains lists that each define a signal for the class. Each list includes the name, return type, and parameter types of the signal. The types may be specified in the same format as used for property definitions. The initialization function, passed as the `.initialize` parameter, is invoked whenever an instance of the class is created, before any properties are set. It takes the newly created instance of the class as its only parameter.

The return value from the call to `gClass` is the identifier of the new `GType`, and this can be used in calls to create instances of this type.

The next step in our example is to create an instance of `RGtkTransformedHScale` and to register a handler on the `value-changed` signal that will draw the plot using the transformed value as the alpha setting:

```
R> adj <- gtkAdjustment(0.5, 0.15, 1.00, 0.05, 0.5, 0)
R> s <- gObject(tform_scale_type, adjustment = adj, expr = expression(x^3))
R> gSignalConnect(s, "value-changed", function(scale) {
+    plot(ma_data, col = rgb(0, 0, 0, scale$getTransformedValue()),
+      xlab = "Replicate 1", ylab = "Replicate 2",
+      main = "Expression levels of WT at time 0",  pch = 19)
+  })
```

Instances of any **GObject** class may be created using the `gObject` function. The value of the `expr` property is set to the R expression $x^3$ when the object is created. The signal handler now calls the new `getTransformedValue` method, instead of `getValue` as in the original version. This final block of code completes the example:

```
R> win <- gtkWindow(show = FALSE)
R> da <- gtkDrawingArea()
R> vbox <- gtkVBox()
R> vbox$packStart(da)
R> vbox$packStart(s, FALSE)
R> win$add(vbox)
R> win$setDefaultSize(400, 400)
R> require("cairoDevice")
R> asCairoDevice(da)
R> win$showAll()
R> par(pty = "s")
R> s$setValue(0.7)
```

More precise details on defining **GObject** classes are available in the R help page for the `gClass` function.

## 6. Language binding design and generation

### 6.1. Goals and scope

There are two primary concerns for the design of **RGtk2**: consistency and efficiency of use. In terms of consistency, the API should be consistent with R first and **GTK+** second. **RGtk2** aims to provide a complete and consistent interface to the **GTK+** API, except where that would conflict with R conventions. This is based on the assumption that the **GTK+** API has been designed to be used as a whole. We purposefully avoid any attempt to limit the bindings to what we might consider the most useful subset of **GTK+**. Only functionality that would introduce foreign concepts to R, such as as memory management, return-by-reference parameters, and type casting, is excluded from the **RGtk2** interface. It should not be obvious to the user that **GTK+** is implemented in a foreign language. As a consequence of consistency with **GTK+**, **RGtk2** provides a fairly low-level interface, which likely detracts from its ease of use. To rectify this, **RGtk2** aims to increase the usability of its API. Towards this end, it provides high-level facilities like the `RGtkDataFrame` utility and the custom syntax for calling methods and accessing properties.

In addition to **GTK+**, **RGtk2** also provides bindings for **Cairo**, **GDK**, **GdkPixbuf**, **Pango**, **ATK**, and **Libglade**. All of these libraries were designed with language bindings in mind, and, except for **Cairo**, they are all based on the **GObject** framework. The API for **Cairo** is sufficiently simple that its independence from **GObject** is of little consequence. As a result, there are no significant binding issues that are particular to a single library, so the discussion of **GTK+** suffices for all of the bindings.

With the exception of properties and signals, which are bound at runtime using introspection, the **RGtk2** bindings, including functions, methods, fields, virtual functions, callbacks and enumerations, are based on programmatically generated code connecting R and the C routines and data structures. This section continues by detailing the code generation system and the type conversion routines utilized by the generated code. It concludes by introducing the system for autogenerating the R documentation for the package. The explanations assume the reader has a working knowledge of the **GObject** system (see Section 5.2).

## 6.2. Automatic binding generation

Given the broad scope of the project, it was decided that developing a system for automatically generating the interface would be more time efficient than manual implementation. Autogeneration also enhances the maintainability of the project, since improved code can be uniformly and programmatically generated across for new versions of each library. Additionally, this allows us and other users to programmatically generate interfaces to other libraries. This section describes the design of the code generation system, beginning with the input format and then explaining how each component of the bindings is generated.

### *The defs format*

The **GTK+** API and other **GObject**-based API's are often described by a Scheme-based (Kelsey *et al.* 1998) format called defs. A defs file describes the types and functions of an API. The autogeneration system for the **RGtk2** bindings takes defs files as its input. This section briefly describes the defs format and how it is leveraged by **RGtk2**. It concludes with a discussion of alternative API description methods.

The defs format supports six different kinds of types: objects, interfaces, boxed types, enumerations, flags and pointers. Each of these correspond to a fundamental `GType` (see Section 5.2).

```
(define-object Widget
  (in-module "Gtk")
  (parent "GtkObject")
  (c-name "GtkWidget")
  (gtype-id "GTK_TYPE_WIDGET")
  (fields
    '("GtkStyle*" "style")
    '("GtkRequisition" "requisition")
    '("GtkAllocation" "allocation")
    '("GdkWindow*" "window")
    '("GtkWidget*" "parent")
  )
)
```

Table 1: An example of the defs format for specifying the API of **GObject**-based libraries. This particular expression describes the `GtkWidget` class.

Every type of definition has a field identifying the module in which it is contained (usually the name of the library or API), its C symbol and its `GType`, with the exception of raw pointer types, which lack a specific `GType`. The objects, boxes, and pointers may contain a list of field definitions, each consisting of the type and name of a field. The type names are formatted as they are in C except for some special syntax for indicating arrays and specifying the type of the elements in a list. Object definitions have a field for the parent type, while definitions of boxed types specify the copy and free functions of the type. Each enumeration and flag definition contains a list of their allowed values. As an example, the defs representation of the `GtkWidget` object is given in Table 1.

In addition to types, the defs format supports definition of four kinds of invocable or callable elements: functions, methods, virtual functions and callbacks. All callable definitions contain the C symbol, a return type, whether the caller owns the returned memory and a list of parameter definitions.

Each parameter definition contains a type, name, parameter direction (in or out), optional default value and optional deprecation message. Parameter direction refers to whether a parameter is passed as input (*in*) to the function or is part of the return value (*out*), which is known as *return by reference* in C. An example is retrieving the dimensions of a `GdkDrawable`, the rectangular target of **GDK** drawing operations. The method `gdkDrawableGetSize` has two integer out parameters, `width` and `height`. A few parameters in the bound API's are sent in both directions, but these so-called *inout* parameters are so rare that we handle them manually. Parameter types are formatted like field types.

There are slight differences in the way the different types of callables are defined in the defs. Functions may be marked as constructors, i.e., for creating objects of a specified type. Methods and virtual functions belong to an object or interface type. This distinguishes them from plain functions and callback functions, which are independent of a class. The name of the type declaring the method or virtual function is specified in the definition. Below is an example of the `getSize` method on `GtkWindow`:

```
(define-method get_size
  (of-object "GtkWindow")
  (c-name "gtk_window_get_size")
  (return-type "none")
  (parameters
    '("gint*" "width" (out))
    '("gint*" "height" (out))
  )
)
```

The Python binding to **GTK+**, **PyGTK** (Chapman and Kelley 2000), provides Python classes for the generation and parsing of defs files. The generation scripts scan C header files for information about an API. The autogenerated defs file is then manually annotated with information that is not derivable from header files, such as that regarding memory ownership. **PyGTK** maintains a set of reference defs files for every library bound by **RGtk2** except **Cairo**, for which a defs description was created as part of this work.

**RGtk2** leverages this information as input to its binding generation system. The system is implemented in R and calls the **PyGTK** defs parsing code via the **RSPython** (Temple Lang

2005b) package. The resulting descriptions are converted to R and from these the interface code is generated, consisting of both R and C binding code. In the great majority of cases, the information provided by a defs file is sufficient for autogeneration of bindings. However, there is a small number of functions that require manual implementation, such as those with variadic arguments or complicated memory ownership policies.

There are some alternatives to the defs format. The **GTK#** project (Bernstein Niel 2004), which binds **GTK+** to the .NET platform, has defined the XML-based GAPI format (The Mono Project 2008). GAPI contains essentially the same information as defs files, but the GAPI tools allow the raw API description, which is normally derived automatically from the header files of the library, to be stored separately from the manual annotations. The raw definitions and annotations are merged when generating the code for an interface. This facilitates maintenance of the interface definitions. The defs tools from **PyGTK** do not support this, although filtering using regular expressions and storing the changes as *patch* or difference files works fairly well. GAPI came long after the introduction of **RGtk**, and it was decided that there were not enough advantages over defs to justify a switch. A second XML-based format, GIR (Clasen *et al.* 2010), has recently been developed as a unifying standard for representing **GObject**-based API's. The use of XML as input to our code generation system would substitute the dependency on the **RSPython** package with the **XML** package, and this might prove simpler.

The following sections introduce each component of the bindings output by the code generation system. The components include a set of wrappers for each callable type, an accessor function for each field, and code for creating an R vector for each enumeration/flags type.

### *Function and method wrappers*

Functions and methods are mapped to R functions of the same name, transformed to *camelBack* form, i.e., words concatenated and the first letter in upper case, except for the first word. Although an object-oriented syntax for methods is supported, its use is not mandatory; every API call is possible through an R function. This results in an interface that is familiar to the R programmer.

Each function and method definition in the defs input is converted to two wrapper functions, one in R and the other in C. The R wrapper is responsible for coercion of the parameters to the R types that correspond to the C types of the parameters of the underlying C function. This includes checking the `class` attribute of the `externalptr` objects for the expected type. It is considered simpler, safer and more maintainable to perform the coercion in R than in C. The R wrapper will optionally emit a warning if the function is deprecated. It then calls the C wrapper for the function, which converts the parameters from R types to C types and invokes the API function. The return value, if any, is converted from C to R. If there are any *out* parameters, these are also converted to R types and bundled with the return value in a list. This avoids the foreign concept of return-by-reference in R. The result is then returned to the R wrapper. If the function is a widget constructor, an extra optional parameter (`show`) is added to the generated R function and this controls whether the newly created widget will immediately be made visible. Finally, the result is returned to the user.

The following is an example of this process for the function `gtkWidgetCreatePangoLayout`, which is a commonly used function for drawing text on a widget, such as a `GtkDrawingArea`. First, we present the autogenerated R wrapper, from the **RGtk2** source code, reformatted to wrap long lines:

```
R> gtkWidgetCreatePangoLayout <- function(object, text) {
+     checkPtrType(object, "GtkWidget")
+     text <- as.character(text)
+     w <- .RGtkCall("S_gtk_widget_create_pango_layout", object, text,
+       PACKAGE = "RGtk2")
+     return(w)
+ }
```

The wrapper ensures that the object is of type `GtkWidget` and coerces the text to display to a character vector. It then invokes the C wrapper with the validated arguments and returns the result. Below is the source code listing of the `S_gtk_widget_create_pango_layout` function:

```
SEXP
S_gtk_widget_create_pango_layout(SEXP s_object, SEXP s_text)
{
  SEXP _result = R_NilValue;
  GtkWidget* object = GTK_WIDGET(getPtrValue(s_object));
  const gchar* text = ((const gchar*)asCString(s_text));

  PangoLayout* ans;

  ans = gtk_widget_create_pango_layout(object, text);

  _result = toRPointerWithFinalizer(ans, "PangoLayout",
    (RPointerFinalizer) g_object_unref);

  return(_result);
}
```

The R types are converted to C types and passed to the actual **GTK+** function. The result, a `PangoLayout` object, is converted to an R `externalptr` type and returned.

### *Constructors*

There are often several constructor routines for a given **GTK+** class, e.g., for `GtkButton` there is `gtkButtonNew`, `gtkButtonNewWithLabel`, `gtkButtonNewFromStock`. While bindings for each of these are available, we also want the R programmer to be able to use a single general purpose constructor function, e.g., `gtkButton()`. Depending on which arguments are provided to `gtkButton()`, the code decides which of the low-level constructors is to be invoked.

For each object class, the high-level constructor function (e.g., `gtkButton()`), is programmatically generated. Its parameter list matches the union of all of the parameter lists for each constructor of the class. The function body delegates to one of the constructors based on which parameters are provided by the user. As with `GtkButton`, the name of the constructor is the name of the class with the first character in lower case.

As an example, the code for the programmatically generated `gtkButton()` function is given below. The `GtkButton` class has three constructors which correspond to the functions `gtkButtonNewFromStock(stock.id)`, `gtkButtonNewWithLabel(label)` and the basic `gtkButtonNew`, which takes no arguments. From these, we generate the following code:

```
R> gtkButton <- function (label, stock.id, show = TRUE) {
+    if (!missing(stock.id)) {
+      gtkButtonNewFromStock(stock.id, show)
+    } else {
+      if (!missing(label)) {
+        gtkButtonNewWithLabel(label, show)
+      } else {
+        gtkButtonNew(show)
+      }
+    }
+  }
```

This then allows the R programmer to use calls such as

```
R> gtkButton()
R> gtkButton("my label")
R> gtkButton(stock.id = "...")
```

### *Callback wrappers*

Callbacks are functions that are passed to and returned from functions and methods in the API of a library. After a callback is registered, the library may invoke it to perform a particular task, and, in so doing, it *calls back* into client code. Thus, callbacks are one means for a client to customize part of the functionality of a library.

An example of a function that registers a callback is `gtkTreeViewColumnSetCellDataFunc`, which is used by the `create_tree_view` function in Section 4.4 to customize the rendering of the numeric values as text in the spreadsheet. There are two integral components of the bindings that enable support for callback functions. First, there is the wrapper for each callback registration function, like `gtkTreeViewColumnSetCellDataFunc`. Then, for each callback function, there is a C implementation of the callback that delegates to the R function registered as the callback by the user.

In general, callback registration functions take two parameters: the callback function and a *user-data* structure that is conventionally passed as the final argument to the callback function. Similar to connecting a handler to a signal, the user needs to provide an R function as the callback function. The *user-data* parameter is optional; if given, it may be any R object.

When wrapping a callback registration function, special handling is required for its parameters. As the underlying C registration function requires a C function for its callback parameter, we pass an autogenerated C function, which delegates to the user-provided R callback function (see below). In order to provide the R function to the C wrapper, we place the R function, as well as the R *user-data* (if any) in a structure and pass that structure to the underlying C registration function as the *user-data*.

The generated code for the C wrapper that delegates to the user-provided R callback is similar to that of an ordinary function wrapper, except the flow of control is in the opposite direction. When invoked, the function retrieves the R function and *user-data* object from the special structure passed as the *user-data* to the wrapper. The wrapper then converts its parameters to their R equivalents, calls the user-provided R function, and returns the result after converting it to its C equivalent.

### *Virtual function wrappers*

Virtual functions, like `format_value` in the `RTransformedHScale` class defined in Section 5.4, are bound to R in order to support the extension of **GObject** classes. Wrappers for virtual functions are generated for both directions, from R to C, like the function wrappers, and from C to R, like callbacks.

Although virtual functions are not public like methods, they are bound in the forward direction for calling the overriden implementation of a virtual function in a parent class. The generated code in this case is very similar to that for wrapping ordinary functions and methods.

The reverse wrapper, from C to R, is needed to support the overriding of virtual functions through inheritance. The R functions implementing the virtual functions are stored within the **GObject** class structure. When a virtual function is invoked, the code searches for a corresponding R function. It is not guaranteed that one exists within the class structure, as a class need not override every virtual function it inherits. If one is found, the R function is invoked in the manner described above for callbacks functions. If no overriding function is found, the code delegates to the implementation of the method in the parent class.

### *Field accessors*

Fields, which are typically considered read-only in **GObject** API's, may be accessed in R using the extraction function `[[` as in `obj[[name]]`. The usage of `[[` is the same as when extracting a named element from an R list, which should be familiar to every R programmer. This mechanism is based on an R wrapper function named according to the scheme *classNameGetFieldName*, e.g., `gtkWidgetGetStyle` for retrieving the `style` field from an instance of `GtkWidget`. This function works much the same as the function bindings introduced above, except the C wrapper accesses a field of a C structure rather than invoking a function, and converts the value from C to R.

### *Enumeration and flag definitions*

Although the function wrappers accept the string representations of enumerations and flags, as that is likely familiar to R programmers, there are some cases, such as in the example in Section 3.1 involving `GtkResponseType` and when performing bitwise operations on flags, that the numeric values of enumerated types are required. The code generator outputs definitions of R numeric vectors with the names corresponding to the string representation of each value.

## 6.3. Type conversion

### *Overview*

Most of the work on **RGtk2** outside of autogeneration deals with type conversion. Conversion of strings and primitive C types, such as `int` and `double`, is relatively obvious and simple. Pointers to C structures are converted in two different ways, generally referred to as *high-level* and *low-level* type conversion. High-level conversion is the translation between a C structure and a native R object, information? such as a list. The class attribute on the object normally corresponds to the type of the original C structure. The alternative is low-level conversion to and from R `externalptr`s. For consistency, the method of conversion is the same for a particular structure type in both directions, to and from C. Collections, such as arrays and linked lists, are converted by iterating over the data structures, converting each element and storing the result into an R list. This section continues with further details on the two methods for converting C structures, and this is followed by explanations of array and error conversion.

### *High-level type conversion*

High-level structure conversion either produces or consumes a native R object instead of a low-level `externalptr`. The advantage of a native R value is more obvious integration with R. In particular, reference semantics are avoided. However, due to performance considerations, information hiding, library design, and other constraints, high-level conversion is only feasible in certain cases. One rare case is where a complex C type has a clear analog in R. An example of this is the `GString` structure, which is a convenience wrapper around an array of characters. This is naturally mapped to an R `character` vector of length 1, i.e., a single string and not a vector of characters. The more common second case is the conversion between C structures and R lists, where each field of the structure is represented by an element in the list, in the same order. The names of the list elements match the names of the structure fields.

Structures qualify for the second case if they are meant to be initialized directly in C and therefore lack a constructor. Although a new function could be introduced as a constructor, this would introduce an unnecessary inconsistency between R and C. In our experience, if the underlying API requires that a structure be initialized directly, it is feasible to perform high-level conversion on the structure. An example of this type of high-level conversion may be found in the spreadsheet example in Section 4. The actions for the menu and tool bar are specified as lists; no external references are created.

### *Low-level type conversion*

The use of low-level `externalptr` objects for the underlying C structures is likely unfamiliar to most R programmers, but, in general, it is difficult to avoid. The primary reason is that the C libraries depend on the treatment of many structures as references. For reasons connected to run-time "safety" and method dispatch, the type of the pointer, as well as the entire class hierarchy in the case of an object, is stored as a character vector in the `class` attribute of the R object. This is used, for example, when validating inputs in function wrappers, as well as for determining the function to call when the user employs the *object-$-method* syntax.

An important consideration when handling references is memory management, which needs to be hidden from the R user. The base policy is that memory is preserved until it is no longer referenced by R. This relies on the R garbage collector and the reference counting of **GObject**. Boxed structures are copied using their copy function and registered for finalization using their free function. Instances derived (directly or indirectly) from the `GObject` class are managed using a reference counting scheme. The reference count is incremented when a reference is obtained and decremented when the reference is finalized. In cases where memory ownership is transferred implictly, such as when an object is constructed, it is not necessary to claim ownership by copying or increasing an reference count.

There are two cases where the above mechanism is insufficient: C structures without `GType`s and objects derived from `GtkObject`, which serves as the base class `GtkWidget`, as well as several other **GTK+** classes. When a structure lacks a `GType`, **RGtk2** does not know how to manage its memory. Thus, the structure is passed to R without copying it or otherwise transferring the ownership of the memory to R, in the hope that the memory is not freed externally. Thankfully, these types of structures are rare. Most of them are converted to high-level R structures, which avoids holding a reference.

The second exception is `GtkObject`, which extends `GObject` to support explicit destruction via the `gtkObjectDestroy` function. When that function is invoked, the `destroy` signal is emitted. All parties that hold a reference to the object are required to respond to the signal by releasing their reference. This functionality is useful for destroying widgets when they are no longer needed, even if other parties hold references to them. However, it also means that the R references to the object will become invalid even though they are still visible to the R session. When a reference to a `GtkObject` is obtained, the **RGtk2** package transparently connects a handler to its `destroy` signal. Besides releasing the reference, the signal handler modifies the `class` attribute of the `externalptr` to a sentinel value indicating that the reference is invalid. If the programmer attempts to use an invalidated reference, an error will be thrown. This silent modification of the class attribute may surprise the R programmer, but it avoids fatal errors that may corrupt the R session (e.g., segmentation faults).

### *Arrays*

C arrays are converted to R lists, with each element converted individually. The primary complication is that C arrays do not track their length. Unless an array is terminated by a sentinel value, there is usually no way to determine the length from the array itself. This requires C functions to accept and return array length parameters along with arrays. Array length parameters need to be hidden from the R programmer, since R vectors have an inherent length. The code generator uses heuristics to identify array length parameters and does not require the R programmer to provide them. For example, if an array parameter is followed by an integer parameter, the generator will assume the integer parameter specifies the length of the array. A specific example of this is the function `gtkActionGroupAddActions`, which is called in Section 4.2 as part of the spreadsheet application example. The function wraps the C function with the signature `gtk_action_group_add_actions(GtkActionGroup *action_group, GtkActionEntry *entries, guint n_entries, gpointer user_data)`. The generator guesses that the unsigned integer `n_entries` parameter represents the length of the array of `GtkActionEntry` structures passed as `entries`. For input parameters, the wrapper passes the length of the input R list as the array length parameter. For returned arrays, a similar heuristic finds the returned length and uses it when converting the array to an R list. For example, the C function `gtk_recent_chooser_get_uris(GtkRecentChooser *chooser, gsize *length)` returns an array of strings, containing the URI's in a recent file chooser widget. The generator identifies the `length` return-by-reference (out) parameter as representing the length of the returned array. This heuristic is slightly less reliable compared to the one for input arrays, since there is no array parameter to search around for a length parameter. In cases where these heuristics fail, we manually implement the function wrapper (relying on the code generator for a head start).

*Errors*

Certain errors that occur in **GLib**-based libraries are described by a returned `GError` structure. In R, the user is often alerted to a problem via a condition emitted by the `stop()` or `warning()` functions. The user may pass a value of `TRUE` or `FALSE` as the value of the `.errwarn` parameter to any wrapper that might raise a `GError`. If `.errwarn` is `TRUE`, a warning is raised. Alternatively, if `.errwarn` is `FALSE`, no warning will be emitted and the user can inspect a returned list structure containing the fields of the `GError`, which often holds more information compared to the warning string. In the future, a new type of R-level condition may be added for a `GError`, but the system currently emits only warnings.

## 6.4. Autogeneration of the documentation

The final design consideration is the documentation of the bindings, which is also accomplished by auto-generation. A relatively easy approach would be to generate a single documentation file with an alias for all of the functions and data structures of a particular library. That file could contain a reference to the library's C documentation on the web. However, referring the user to C documentation would have several disadvantages. First, most R programmers are likely not familiar with C. Second, there would be a number of significant inconsistencies in the API. This might confuse even an experienced C programmer. For example, **RGtk2** hides function parameters that specify the lengths of arrays, since these are always known in R. The existence of these in the C documentation would confuse the R user. Other inconsistencies would be return-by-reference parameters and the names of data types. Also, the C documentation would omit concepts such as high-level structure conversion.

Fortunately, all of the bound libraries rely on the **gtk-doc** utility that produces documentation as Docbook XML. The XML representation may be parsed into R using the **XML** package (Temple Lang 2001). From within R, it is possible to introspect the bindings and access the API descriptions stored in the defs files. By combining this information with the original documentation, the documentation generator is able to output R help files that are consistent with the **RGtk2** API. Embedded C examples are replaced with their R equivalent by looking up an R translation by the name of the example. The translation is done manually. The generator attempts to filter out irrelevant statements, such as those regarding memory management, though many C-specific phrases still exist in the output. Thus, the documentation of **RGtk2** is still very much a work in progress.

# 7. Technical language binding and GUI issues

## 7.1. Fully programmatic binding generation

The strategy of autogenerating the bindings saves a significant amount of time and facilitates maintenance, but it is not without its problems. The defs files as generated from the header files do not contain all of the information necessary to correctly generate bindings to many of the C functions. This requires human annotatation of the defs files. The two most common types of required annotation are the direction of parameters (in, out or inout) and the transfer of memory ownership. There is no way to determine this information from the header files.

Another solution would be to require the authors of the API to include the missing information as specially formatted comments in the source code. The comments could even be part of the inline documentation, as it would be beneficial to state such information in a standard way in the documentation, as well. This method does not avoid human annotation and there is the potential for the code and the documentation to become unsynchronized, but the benefit is that the annotations are centrally maintained by an authoritative source.

A variation on the above idea would be to support registration of functions, with all information necessary for binding, during class initialization, just as signals and properties are currently. This would render the entire API of a library introspectable at runtime; compiled bindings would no longer be necessary. However, runtime introspection of functions would have a high performance cost due to the consumption of a large amount of memory and the need to lookup the information each time it is needed. One way around this would be to use the information for generating a compiled interface but not to load the information during normal use of the library. Still, the previous solution of storing the information in comments would have the advantage of being accessible without linking to the library.

A more radical solution would be to write libraries in an entirely different language, which compiled down to **GObject**-based C code. The design of the language would ensure that all information necessary for binding would be known to the compiler. One such language is named Vala (Billeter and Sandrini 2010). Vala is an object-oriented language with a C# syntax and features like assisted memory management, lambda expressions and exceptions. The Vala compiler provides an API for inspecting the parsed language, from which binding information like memory management and function parameter directions may be obtained.

While completely reimplementing a C library in Vala would likely be impractical, it is possible to write interface stubs in a subset of Vala, keeping the implementation in C. This interface definition language, known as VAPI, could then be translated to a standard format, e.g., GIDL, for input to interface generation tools. In the case of GIDL, this avoids the need for a human to write XML.

All of the above solutions require, to some extent, manual maintenance of the interface defintions. One way the machine might programmatically determine information about return-by-reference parameters, memory management and other aspects would be to inspect the C source code of the library in addition to or instead of the header files. The **RGCCTranslationUnit** package (Temple Lang 2006a) provides a framework and some tools to support such inspection.

## 7.2. RGtk2 as a base for other GObject bindings

Implementations of most progamming languages are still written in C. This suggests that libraries implemented in C are likely accessible to more languages than those implemented in Java, for example. **GObject** is designed with language bindings in mind. Given this incentive, it is likely that the number of **GObject**-based libraries will continue to grow.

**RGtk2** has been designed to serve as a base for other R packages binding to **GObject**-derived libraries. The mechanism introduced by R 2.4 for sharing C interfaces between packages allows **RGtk2** to export all of its C-level utilities for interacting with **GObject**, including type conversion routines, wrappers for the **GObject** API, and functions for extending **GObject** classes. This support has already been used by an experimental version of **rggobi** (Lawrence *et al.* 2007). If this functionality proves to be of general use, it should probably be split out of **RGtk2** as a base binding to **GObject**. In conjunction with this, the binding generation system should be revised and made public, as was done for the original **RGtk** package.

### 7.3. Event loop issues

All user interfaces need to respond to user input. **GTK+** provides an *event loop* that checks for user input and executes application callbacks when necessary. **GTK+** applications written in C usually execute the **GTK+** event loop after initialization. The loop takes is started and continues processing events until the GUI is terminated. The interactive R session is a user interface, and it has its own event loop. When using **RGtk2** from an interactive session, there are two event loops, R and **GTK+**, trying to process the user input at the same time.

By default, **RGtk2** attempts to reconcile the two loops by delegating to the **GTK+** event loop when the R event loop is idle. In general, both interfaces operate as expected under this configuration. However, as the **GTK+** event loop is not iterated continuously, certain operations, in particular timer tasks, are not executed reliably. While it is not expected that many **RGtk2** users will rely on timers, several **GTK+** widgets use timers for animation purposes. These widgets tend not to be as responsive as the others without reliable iteration of the **GTK+** event loop. One solution to this problem is to invoke the R function `gtkMain`, which transfers control to the **GTK+** event loop and blocks the R console for the lifetime of that GUI. If the user is willing to sacrifice access to the regular R console, this is a viable method to enhance the responsiveness of the **GTK+** GUI. Of course, an alternative command line interface can be provided by implementing it within a **GTK+**-based GUI and so the user would have both. Indeed, we feel that R should not have its own event loop but rather be treated as library from other front ends. Another possible solution would be a multithreaded model, with synchronized access to the R evaluator. There has been some work towards a solution to this problem, such as the **REventLoop** package (Temple Lang 2003), but this remains an area for further research.

## 8. Comparison of RGtk2 to other R GUI toolkit bindings

There are many different ways to construct a GUI from R. All of them, at some level, depend on a binding to an external widget toolkit. Direct bindings exist for Tcl/Tk (Ousterhout 1994; Welch 2003) and **wxWidgets** (Smart *et al.* 2005), in addition to **GTK+**. Other toolkits are indirectly accessible across interfaces to **DCOM** (Microsoft Corporation 2007) and Java (Sun Microsystems 2007). This section outlines the alternatives to **RGtk2** for constructing GUIs in R, considering the features of both the R binding and the underlying toolkit.

The great majority of R GUIs rely on the **tcltk** package (Dalgaard 2001, 2002) that binds R to Tcl/Tk (Ousterhout 1994; Welch 2003), a mature light-weight cross-platform widget library. Applications of **tcltk** range from **limmaGUI** (Smyth 2005), a task-specific GUI for microarray preprocessing, to the more general R Commander (Fox 2005). The **tcltk** package is bundled with the core distribution of R. This means that developers can usually count on its availability. This is not the case for **RGtk2**, which requires the user to install **RGtk2**, **GTK+**, and all of the libraries on which **GTK+** depends. The small footprint of Tcl/Tk likely delivers better performance in terms of speed and memory than **GTK+** in many circumstances. Tcl/Tk also offers some features that base **GTK+** currently lacks, the canvas widget being one example.

Unfortunately, Tcl/Tk development is slow and the library is beginning to show its age. It lacks many of the widgets present in **GTK+** and other modern toolkits, such as tree tables, progress bars, and autocompleting text fields. Tcl/Tk widgets are often less modern or sophisticated than their **GTK+** counterparts. For example, a **GTK+** menu is able to be torn off as an independent window and the **GTK+** file chooser supports the storage of shortcuts. Tcl/Tk also lacks theme support, so it is not able to emulate native look and feels. Tcl/Tk is not object-oriented, and it is not possible to override the fundamental behavior of widgets. While one can build so-called "megawidgets" on top of existing **Tk** widgets, this is not the same as creating new `GtkWidget`-derived classes with **RGtk2**. Moreover, the design goals of the **tcltk** package differ from those of **RGtk2**, in that **tcltk** aims to expose the functionality of the Tcl engine to the R programmer, while **RGtk2** is a higher-level binding to a collection of specialized C libraries.

The Windows-specific **tcltk2** package (Grosjean 2010) is an attempt to overcome some of the limitations of the **tcltk** package by binding the **Tile** extension (English 2004) of Tcl/Tk. Tile adds support for themes, allowing emulation of native widgets and prettier GUIs, as well as new widgets like a tree table and progress bar. However, Tile still lags behind **GTK+**. For example, the **GTK+** tree table allows the embedding of images, check boxes, and combo boxes, while the **Tile** one does not.

**wxWidgets** (Smart *et al.* 2005) differs from Tcl/Tk and **GTK+** in that it provides a common API with platform-specific implementations based on the native widgets of each platform, and so preserves the look and feel of each platform, without resorting to emulation. In contrast, Tcl/Tk and **GTK+** provide exactly the same widgets on all platforms, leaving the look and feel to theme engines. **GTK+** serves as the "native" Linux implementation of **wxWidgets**. The first binding from R to **wxWidgets** is the now defunct **wxPython** package that leverages **RSPython** to access the Python binding to **wxWidgets**. **RwxWidgets** (Temple Lang 2008b) is a more recent binding that directly binds to the C++ classes of **wxWidgets**. **wxWidgets** has some limitations; it lacks integrated 2D vector graphics and support for constructing GUIs from XML descriptions. However, **wxWidgets** does provide some features that do not exist yet in base **GTK+**, such as HTML display and a dockable window framework.

The **RDCOM** (Temple Lang 2008a) and **rcom** (Baier 2009) packages provide an interface between R and **DCOM** (Microsoft Corporation 2007). This permits manipulation of existing GUIs, such as that of Microsoft Office, or programmatically placing ActiveX controls on an Excel spreadsheet and, with the **RDCOMEvents** package (Temple Lang 2005a), connecting R functions to their events. The **R-(D)COM** package has been used to create the educational R GUI **simpleR** (Maier 2006). A major drawback to the use of **DCOM** is its dependence on Microsoft Windows. However, given the prevalence of Microsoft Windows, this is a significant benefit for those seeking to develop rich GUIs for that platform and integrating tools such as Excel, Word and Internet Explorer.

Java toolkits, including **Swing** and **SWT**, are also accessible from R through R-Java interfaces such **rJava** (Urbanek 2009). The features of **Swing** and **SWT** are comparable to those of **GTK+**, and one could use **rJava** to develop Java-based GUIs. This would be facilitated by a high-level interface for GUI development built on top of the low-level interface provided by **rJava**.

Such an interface is delivered by the **gWidgets** package (Verzani 2009). **gWidgets** provides a simplified, common-denominator-style API for GUI programming that, similar in spirit but not as complete as the approach of **wxWidgets**, is implemented by multiple toolkit backends. **gWidgets** is written in R, so its backends rely on bindings to the external toolkits. So far, there are three backends for **gWidgets**: **gWidgetsRGtk2**, based on **RGtk2**; **gWidgetsJava**, based on **rJava** and **Swing**; and **gWidgetsTcltk** for Tcl/Tk. A defining characteristic of **gWidgets** is the design of its API, which aims for simplicity and consistency with R conventions. The goal is to accelerate the construction of simple GUIs by those inexperienced with GUI programming. For this purpose, using **gWidgets** is likely a better course than direct use of **RGtk2**; however, the simplified interface hides functionality that more complex applications might find useful.

The **Qt** toolkit (Blanchette and Summerfield 2008) is another widely used C++ library for developing GUIs. It is used as the basis for the **KDE** desktop and applications (KDE Foundation 2010) and provides a rich, high-quality collection of customizable and extensible widgets, along with many supporting classes for general computing such as network access and threads. It is developed by the company Trolltech and was originally a commercial offering. Subsequently, it has been released with an open source license. The most recent version (4.4) includes the integration of **WebKit** (The WebKit Open Source Project 2010), the libraries used in Apple's **Safari** Web browser, and so offers a wide variety of high quality support for rendering HTML, executing JavaScript, XPath, etc. Bindings for R to **Qt** are in preparation (Lawrence and Sarkar 2010).

One benefit of **RGtk2** (and **RwxWidgets** on Linux) is the capability to integrate with other GUIs based on **GTK+**. Such software includes **GGobi**, **Mozilla Firefox** (on some platforms), and **Gnumeric**. Widgets from these tools could be embedded in **RGtk2**-based GUIs. The **rggobi** package enables this for **GGobi**, a software tool for multivariate graphics.

## 9. Impact and future work

**RGtk2** aims to provide a consistent and efficient interface to **GTK+** for constructing GUIs in R. The design of the API prioritizes usability from the perspective of the R programmer. The package has been adopted by several projects, including: **gWidgets** (Verzani 2009), a simple interface for GUI construction in R; **Rattle** (Williams 2010), a data mining GUI based on **Libglade**; and **playwith** (Andrews 2010), a package for interactive R graphics. Future plans for **RGtk2** include more fully automating the code generation process and keeping pace with frequent **GTK+** releases.

## Acknowledgments

Michael Lawrence's work was supported in part by National Science Foundation Arabidopsis 2010 grants DBI-0209809 and DBI-052067. We also thank Dr. Dianne Cook for providing helpful feedback on the software and this paper.

## References

Andrews F (2010). ***playwith**: A GUI for interactive plots using **GTK+***. R package version 0.9-45, URL `http://CRAN.R-project.org/package=playwith`.

Baier T (2009). ***rcom**: R **COM** Client Interface and Internal **COM** Server*. R package version 2.2-1, URL `http://CRAN.R-project.org/package=rcom`.

Bernstein Niel M (2004). *Using the **Gtk** Toolkit with **Mono**.* URL http://ondotnet.com/pub/a/dotnet/2004/08/09/gtk_mono.htm.

Billeter J, Sandrini R (2010). "Vala – Compiler for the **GObject** Type System." Version 0.7-10, URL http://live.gnome.org/Vala.

Blanchette J, Summerfield M (2008). *C++ GUI Programming with **Qt4**.* Prentice Hall Open Source Software Developer Series, 2nd edition. Prentice Hall, Upper Saddle River, NJ, USA.

Breuer H, Clasen M, Lillqvist T, Janik T, Pennington H, Steinke R, Taylor O, Wilhelmi S (2010). ***GLib**: Low-Level Core Run-Time Library.* Version 2.23, URL http://developer.gnome.org/doc/API/2.0/glib/.

Chapman M, Kelley B (2000). "Examining the **PyGtk** Toolkit." *Dr. Dobb's Journal of Software Tools*, **25**(4), 82.

Clasen M, Dahlin J, Billeter J, Van Hoof P, Taylor R (2010). ***GObject** Introspection.* Version 0.6, URL http://live.gnome.org/GObjectIntrospection.

Cook D, Lawrence M, Lee EK, Babka H, Wurtele ES (2008). "**explorase**: Multivariate Exploratory Analysis and Visualization for Systems Biology." *Journal of Statistical Software*, **25**(9), 1–23. URL http://www.jstatsoft.org/v25/i09/.

Dalgaard P (2001). "A Primer on the R-**Tcl/Tk** Package." *R News*, **1**(3), 27–31. URL http://CRAN.R-project.org/doc/Rnews/.

Dalgaard P (2002). "Changes to the R-**Tcl/Tk** Package." *R News*, **2**(3), 25–27. URL http://CRAN.R-project.org/doc/Rnews/.

Drake L, Plummer M, Temple Lang D (2005). ***gtkDevice**: Loadable and Embeddable **Gtk** Device Driver for R.* R package version 1.9-4, URL http://CRAN.R-project.org/src/contrib/Archive/gtkDevice/.

English J (2004). "The Tile Widget Set." *Tcl2004*. URL http://tktable.sourceforge.net/tile/tile-tcl2004.pdf.

Fourdan O (2000). "**Xfce**: A Lightweight Desktop Environment." In *Proceedings of the 4th Annual Linux Showcase, Atlanta.* Atlanta, Georgia. URL http://www.usenix.org/publications/library/proceedings/als00/2000papers/papers/full_papers/fourdan/fourdan.pdf.

Fox J (2005). "The R Commander: A Basic Statistics Graphical User Interface to R." *Journal of Statistical Software*, **14**(9), 1–42. URL http://www.jstatsoft.org/v14/i09/.

Grosjean P (2010). ***tcltk2**: Tcl/Tk Additions.* R package version 1.1-2, URL http://CRAN.R-project.org/package=tcltk2.

**GTK+** Development Team (2010). ***GTK+**: The Gimp Toolkit.* Version 2.19, URL http://www.gtk.org/.

Helbig M, Theus M, Urbanek S (2005). "**JGR**: Java GUI to R." *Statistical Computing and Graphics Newsletter*, **16**(2), 9–12. URL http://stat-computing.org/newsletter/issues/scgn-16-2.pdf.

KDE Foundation (2010). *K Desktop Environment.* Software version 4.4, URL http://www.kde.org/.

Kelsey R, Clinger W, Rees J (1998). "Revised Report on the Algorithmic Language Scheme." *ACM SIGPLAN Notices*, **33**(9), 26–76.

Krause A (2007). *Foundations of **GTK+** Development.* Apress, Berkely, CA, USA.

Lawrence M (2009). ***cairoDevice**: **Cairo**-based Cross-Platform Antialiased Graphics Device Driver.* R package version 2.10.0, URL http://CRAN.R-project.org/package=cairoDevice.

Lawrence M, Sarkar D (2010). ***qtbase**: Interface between R and **Qt**.* R package version 0.6-3, URL http://qtinterfaces.R-Forge.R-project.org/.

Lawrence M, Wickham H, Cook D (2007). "**GGobi** Beta Homepage." URL http://www.ggobi.org/beta/.

Maier G (2006). ***simpleR**: A Windows GUI for R.* R package version 0.1.1, URL http://www-sre.wu.ac.at/SimpleR/.

Microsoft Corporation (2007). "Distributed Component Object Model (DCOM) for Windows 98." URL http://www.microsoft.com/com/.

Ousterhout J (1994). *Tcl and the **Tk** Toolkit.* Addison-Wesley, Reading, MA, USA.

Penners R (2005). ***GTK-WIMP** (Windows Impersonator).* Version 0.7.0, URL http://gtk-wimp.sourceforge.net/.

R Development Core Team (2010). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

SAS Institute Inc (2007). ***JMP** 7: Statistical Discovery Software.* Cary, NC. URL http://www.jmp.com/.

Smart J, Hock K, Csomor S (2005). *Cross-Platform GUI Programming with **wxWidgets**.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

Smyth GK (2005). "**Limma**: Linear Models for Microarray Data." In R Gentleman, V Carey, S Dudoit, R Irizarry, W Huber (eds.), *Bioinformatics and Computational Biology Solutions Using R and **Bioconductor***, pp. 397–420. Springer-Verlag, New York.

Sun Microsystems (2007). "Java." URL http://www.java.com/.

Temple Lang D (2001). "Using XML for Statistics: The **XML** Package." *R News*, **1**(1), 24–27. URL http://CRAN.R-project.org/doc/Rnews/.

Temple Lang D (2001-2005). ***RGtk**.* R package version 0.8-0, URL http://www.omegahat.org/RGtk/.

Temple Lang D (2003). ***REventLoop**.* R package version 0.2-3, URL http://www.omegahat.org/REventLoop/.

Temple Lang D (2005a). ***RDCOMEvents***. R package version 0.3-1, URL http://www.omegahat.org/RDCOMEvents/.

Temple Lang D (2005b). ***RSPython**: R/SPlus – Python Interface.* R package version 0.7-1, URL http://www.omegahat.org/RSPython/.

Temple Lang D (2006a). ***RGCCTranslationUnit***. URL http://www.omegahat.org/RGCCTranslationUnit/.

Temple Lang D (2006b). ***SJava**: The R & S Interface to Omegahat and Java.* R package version 0.69-0, URL http://www.omegahat.org/RSJava/.

Temple Lang D (2008a). ***RDCOM** Bundle of Packages.* Version 0.92-0, URL http://www.omegahat.org/RDCOMBundle.

Temple Lang D (2008b). ***RwxWidgets***. R package version 0.5-7, URL http://www.omegahat.org/RwxWidgets/.

Temple Lang D, Swayne DF (2001). "**GGobi** Meets R: An Extensible Environment for Interactive Dynamic Data Visualization." In K Hornik, F Leisch (eds.), *Proceedings of the 2nd International Workshop on Distributed Statistical Computing, March 15–17, 2001, Technische Universität Wien, Vienna, Austria.* ISSN 1609-395X, URL http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/.

The Cairo Project (2010). ***Cairo** Vector Graphics Library.* Version 1.8.10, URL http://www.cairographics.org.

The Mono Project (2008). "GAPI." URL http://www.mono-project.com/GAPI.

The WebKit Open Source Project (2010). ***WebKit***. URL http://webkit.org/.

Tierney L (2008). ***tkrplot**: **Tk** R Plot.* R package version 0.0-18, URL http://CRAN.R-project.org/package=tkrplot.

Unwin A, Hofmann H (1999). "GUI and Command-line – Conflict or Synergy?" In K Berk, M Pourahmadi (eds.), *Computing Science and Statistics.*

Urbanek S (2009). ***rJava**: Low-Level R to Java Interface.* R package version 0.8-1, URL http://CRAN.R-project.org/package=rJava.

Verzani J (2009). ***gWidgets**: **gWidgets** API for Building Toolkit-independent, Interactive GUIs.* R package version 0.0-37, URL http://CRAN.R-project.org/package=gWidgets.

Verzani J (2010). ***pmg**: Poor Man's GUI.* R package version 0.9-42, URL http://CRAN.R-project.org/package=pmg.

Walthinsen E (2001). "**GStreamer** – GNOME Goes Multimedia." *Technical report*, GUADEC.

Warkus M (2004). *The Official GNOME 2 Developer's Guide.* 1st edition. No Starch Press, San Francisco, CA, USA.

Welch B (2003). *Practical Programming in Tcl and **Tk**.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

Williams G (2010). ***Rattle**: GNOME R Data Mining.* R package version 2.5.24, URL http://CRAN.R-project.org/package=rattle.

**Affiliation:**

Michael Lawrence
Bioinformatics and Computational Biology
Genentech Research and Early Development
South San Francisco, CA, United States of America
E-mail: michafla@gene.com

*Journal of Statistical Software* http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 37, Issue 8 *Submitted:* 2009-01-30
December 2010 *Accepted:* 2009-09-23