RGtk2 Overview

Michael Lawrence and Duncan Temple Lang

January 31, 2006

1 Introduction

RGtk2 provides R with access to a large collection of related libraries: GLib, GObject, Pango, ATK, GDK, GTK, Cairo, Libglade, and (depending on the platform) GtkMozEmbed. GTK is of course the primary binding, which all the other bindings are meant to support. GTK is a toolkit for creating graphical user interfaces. It provides two basic types of interaction.

Widgets A large collection of components that can be used to create the GUI. These include common primitive elements such as buttons and labels, menu items, text widgets, drawing areas, top-level windows, . . . with which one can make more complex composite widgets such as dialogs, calendars, file selection interfaces, etc. In addition to the low-level, action widgets, there are also container widgets whose task is to houand manage other widgets. The different types of container widgets manage the space they provide to their child widgets in different ways to give different visual effects, specifically when a window/widget is resized. Container widgets include notebooks with tabs for each separate page, scrolled windows which provide horizontal and vertical scrollbars that get associated with and control the visibility of a child widget; different packing widgets such as a table, a box, a menu and menu bar...

Callbacks Also, Gtk provides a way to associate handlers or actions with particular events on these different components so that one can give the GUI its behavior. In the case of R, these callbacks are given primarily as S functions. These are called when the event occurs with arguments that identify the details of the event, including the particular widget in which the event happened.

When developing a GUI, typically one first creates the visible part, i.e. the collection of different widgets. We do this by creating instances of the different Gtk widget classes, creating the basic elements and "adding" them to the desired container widget. Having created the elements, we then display or "show" the top-level element, be it a top-level window or merely a container to be added to an existing top-level container. Please see the RGtk2 documentation for more details about available widgets.

Having created the physical display for the GUI by creating and arranging the different widgets, we next register the different callback functions with the particular widgets and specifically with the different events of interest. Again, one must learn which events are associated by which type of widget, and when and how the handler will be called. One can use the RGtk2 documentation for this purpose. In general, each callback function will be invoked with at least one argument: the object in which the event occurred. Callbacks for different events may provide additional arguments which provide more information about the particulars of the event. For example, when a button is released in a widget, the button-release event passes the widget and also a GdkEventButton instance which gives information about which button was released, etc.

In addition to the arguments provided by Gtk, one can also associate an S object with a widget and event and have this passed to the callback function as an argument. By associating different objects with different widgets, one can use the same callback function. That function can implement different behavior based on the additional argument, and using R's lexical scoping one can even modify the S object passed as an additional argument.

In the next few sections, we will describe how we can implement the very basic and often-used Hello World to illustrate the essential concepts in the RGtk2 package. This is a very simple GUI which presents a button in its own window. When the user clicks the button, we print a message on the console.

2   Creating GTK Objects

As we saw above, one starts creating a GUI by instantiating different GTK objects. A GTK object is derived from the generic *GObject*. The GObject API is mostly hidden from the RGtk user, except for those functions involved in registering callbacks against object "signals" (user events in the case of GTK objects) and getting/setting object properties (less frequently used).

In the case of the "Hello world" application, we need to create i) the top-level window, and ii) the button which the user clicks. We create the window using the *gtkWindow()* function

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

This creates an instance of the *GtkWindow* class. Generally, the S language constructor function for a Gtk class named *Gtk<Class>* is given by *gtk<Class>()*, i.e. replace the capital G starting the word with a lower case "g".

Note that the constructor functions for each class that extends *GtkWidget* have an optional show argument. This controls whether the widget is made ready for showing immediately or if this must be done by the programmer at a later time. The advantage of deferring this is usually a marginal gain in efficiency. Hence, the default is *TRUE*. One need only prohibit the top-level container, e.g. the window in this example, from being shown and then none of the sub-widgets will be displayed.

We can invoke methods on the Gtk objects to query or modify their state. For example, we can set the title for the frame of the window using the underlying C-level routine *gtk_window_set_title()* provided by the Gtk libraries. We do this in S via the command

```
gtkWindowSetTitle(win, "Hello world test")
```

There are several things to note here. Firstly, we use a different naming convention than Gtk's C-level API. Specifically, we eliminate the underscores (_) and capitalize all but the first word (i.e. the next letter after the _). Secondly, we pass the Gtk object on which we are operating as the first argument. Thirdly, the type of the second argument is defined by the underlying C routine and is a string. This corresponds to a character vector of length 1 in S.

The case of *gtkWindowSetTitle()* is quite simple. We started with an object of class *GtkWindow* in R (created using the S constructor) and then invoked the function gtkWindowSetTitle() for that same class. But what about, for example, the general functions to show or hide a widget, get its parent widget, etc. These apply to all *GtkWidget* objects, and not just the *GtkWindow* objects. Accordingly, the S interface uses the names that correspond to the C-level API and are prefixed by *gtkWidget...()*, rather than *gtkWindow...()*. This makes it hard to remember the precise name of the function one wants to call since it depends on the inheritance or class hierarchy of the Gtk classes.

To make things simpler, we allow one to use a more Java/C++ style that allows users to invoke methods on an object and leave the S engine to determine the precise name to use. Specifically, we use the $ operator on the object followed by the name of the method to identify the function. Specifically, one can invoke from S a method on an underlying Gtk object, say *g* using the form

```
g$MethodName(arg1, arg2)
```

This eliminates the need to remember for which class the method is defined and hence the prefix. Also this form of invocations inserts the target object, *g*, as the first argument in the call to the real S function being called and so reduces typing.

An example will make things clear. Consider again setting the title of the window. Rather than using *gtkWindowSetTitle()*, we can use the command

```
win$SetTitle("Hello world test")
```

This looks for the appropriate function given the class and parent classes of *win* and then invokes the "nearest" function. This corresponds to the command

```
gtkWindowSetTitle(win, "Hello world test")
```

above, but is easier for the user and is also more robust to changes in the class hierarchy and C-leve API.

There is a marginal penalty in computational performance, but this may disappear in the future and is also not likely to be a serious issue a) when creating the GUI and b) given the overhead in setting up callbacks to S functions.

We can now continue with our "Hello world" example. We have created the window and set its title and hence seen how to create Gtk objects and invoke methods. And so creating the button becomes quite simple. We choose the appropriate Gtk class - *GtkButton* - and find the appropriate constructor.

There are two constructors in the C-level API for this class: one that takes no arguments and another that takes a string to display as the text in the button. In S, these two constructors map to a single constructor function, whose name is the name of the class suitably (de-)capitalized, *gtkButton()*. If one calls it with no arguments, the first C-level constructor is called. Alternatively, if one gives a character vector of length 1 as the first argument, the second version is called. More generally, the R interface to Gtk attempts to map the constructor routines into a single S function that can determine which C routine to call based on the number and/or type of the arguments. For the most part, this is quite simple and works effectively.

In our example, we specify text for the button's display and so call

```
btn <- gtkButton("Say 'Hello World'")
```

Next we put the button into the top-level window. The latter is a *GtkContainer* object and has a default mechanism for placing children widgets. Since this is the only widget we will display in the window, we don't have to worry about how to aportion space between different widgets, etc. All we need do is invoke the *add()* method on the window, giving it the child widget which is the button.

```
win$add(b)
```

When we create the button, we did not provide a value for the show argument and so the button is potentially visible. To actually see it, however, we need to show the top-level widget, i.e. the window. We do this by explicitly calling its *show()* method.

```
win$Show()
```

3 Callbacks

At this point, we have created a Gtk GUI that one can see on the screen and can even interact with by clicking on the button. The next step is to make it do something when we click on the button, and this is where we look at callbacks.

The usual types of events are user interactions such as clicking on a button, dragging the thumb of a slider, moving the mouse over a drawing area, etc. Other types of events might be less visible and more abstract such as text being pushed or popped onto a status bar, a new data set being created, and so on. Basically, each type of event is associated with a Gtk object in which it "occurs".

A Gtk object can support different types of events, and events in different objects are treated independently. One creates and customizes an application by connecting different pieces of code that are executed when particular Gtk objects raise/emits particular events.

In our example, we want to execute a simple piece of S code that is executed when the user clicks on the button. The code simply writes the string "Saying hello from the button" to the console via the *cat()* function. To arrange this, we can look at the different signals that the button supports. (Of course, we chose the *GtkButton* class because it provided the appropriate signal, so this seems like we are going around in circles. In general, knowing the widget to use and appropriate signal is the trick to using any toolkit.)

Using the help pages for RGtk, we can find out that the button supports 6 diffferent types of signals itself, and inherits many others from its ancestor classes (*GtkBin, GtkContainer, GtkWidget, GtkObject*, and *GObject*). These signal names are *activate, pressed, released, clicked, enter* and *leave.* The one we are interested in is clicked. We specify our callback for the particular button using the method *gSignalConnect()*. We specify the name of the signal (i.e. *clicked*) and an invokable S object which will be called when the signal occurs:

```
gSignalConnect(btn, "clicked", quote(cat("Saying hello from the button\n")))
```

Now, when you click on the button, the string will be printed on the console. The code that is to be called when the event occurs can be an S expression or call, or a function. If it is an expression or call, then it is evaluated when the event occurs. One typically creates such callable objects using quote(), expression() or substitute(). Each of these types of callbacks is evaluated as a toplevel expression and one is presumably interested in its global side effects, such as changing the value of a session-wide variable, writing to a file or the console, or updating one or more graphics devices.

If the callback is a function, then it is invoked slightly differently. There is more information available to the callback, specifically, the arguments that are made available at the C level by the Gtk API. These are passed onto the S function. This collection of arguments always includes the Gtk object for which the signal is being emitted. Many signals also provide additional values that parameterize the event and allow the callback to be written generally but parameterized by the widget or other event-specific values. These values are converted to S objects using the basic conversion mechanism. In addition to the event-specific values passed from Gtk, one can also specify S objects that Gtk remembers and passes to the function when it is called. Again, this allows one to parameterize general S functions to act on the specifics of the event.

Note that we added the callback after the button was created and visible. This is not necessary and we can add it before the top-level window is shown. However, it does illustrate that we can dynamically add callbacks at any time. Indeed, we can add multiple callbacks to the same Gtk object, and even for the same signal. For example, let's add a second that prints "And again".

```
        id <- gSignalConnect(btn, "clicked", quote(cat("And again\n")))
```

Go ahead and click on the button now and see that two lines of output are produced.

And, of course, if we can dynamically add callbacks, we must also be able to remove them at any time. To do this, we use the *gSignalHandlerDisconnect()* method for the *GObject*. We give it the identifier for the registered callback that we received from *gSignalConnect*. So to un-register the second callback, we issue the S command

```
        gSignalHandlerDisconnect(btn, id)
```

Again, click on the button and you should get only one line of output, specifically saying "hello" from the button.

4   Intermediate Concepts

4.1   Enumerations and Flags

Enumerated types and flags are symbolic constants that are used to identify different states or combinations of states. In R, we represent these as named integers. The intent is that the user will provide the name (or names for flags) and not a simple integer value. So, for example, when specifying the type of window in a call to *gtkWindowNew()* we can use any of the names from the *GtkWindowType* vector representing the
enumeration:

```
    > GtkWindowType
    toplevel dialog popup
        0        1      2
```

Since this is an enumeration, we specify just one of these values, as in

```
    > gtkWindowNew("toplevel")
```

When a flag value is expected, we can combine different values together. Since we can OR (|) names together, we need an alternative syntax. For this, we use a simple character vector containing the names of the flag elements.

As an example, consider the display options for controlling the appearance of the calendar widget. The *GtkCalendarDisplayOptions* is a named integer vector giving the different names for the flag values. If we want to have weeks start on a monday and also show week numbers, we can do this as

```
    > gtkCalendarDisplayOptions(cal, c("week-start-monday","show-week-numbers"))
```

To activate all options, we can use

6

```
gtkCalendarDisplayOptions(cal, names(GtkCalendarDisplayOptions))
```

The calendar can be create and display using the following code

```
cal <- gtkCalendar()
gtkCalendarDisplayOptions(cal, c("week-start-monday","show-week-numbers"))
w <- gtkWindow()
w$Add(cal)
```

Using names guarantees the validity of the value as it is resolved and checked
at run time. However, to guard against erroneous values, we have C-level code
that checks an integer value is within the appropriate set of C-level values and
returns an object representing that symbolic value.

One can note the fact that the name toplevel is converted to $GTK\_WINDOW\_TOPLEVEL$
in the value returned by the enumeraton. This is the C-level name for the enu-
meration. It can be used as a synonym for the value. In other words, *toplevel*
and $GTK\_WINDOW\_TOPLEVEL$ are the same. And indeed, for every enu-
meration or flag we have both a sets of element names available. The local
version is available as described above by giving the name of type, e.g. *GtkWin-
dowType* and *GtkCalendarDisplayOptions*. Prefixing the name with a . gives
the alternative version with the longer, internal names. Use whichever form you
desire. Those who write Gtk code in other languages may be familiar with the
internal names. The local names are shorter.

4.2   Accessing Object Properties

Each *GObject* instance supports values that are accessible by name. The col-
lection of properties can be accessed via the *names()* function and this makes
the object look like a list of named values. Properties are inherited through the
type hierarchy.

Each property has a specific type that can be assigned to it. Some of these
values are are writeable, while most are readable. Additionally, the collection
for a given instance is made up of combining the properties from the different
classes from which the instance is derived. One can discover all this information
using the function *gObjectGetPropInfo()*.

```
b <- gtkButton("Some text")
names(b)
b["label"]
b["label"] <- "A Replacement string"
gObjectGetPropInfo(b)
```

Any function in RGtk2 that requests property value pairs should be passed
property values as arguments with the property names as the argument names.

4.3   Timers & Idle Tasks

The RGtk2 binding to the GLib library allows one to interact with the core event loop (*GMainLoop*) that drives GTK. *gTimeoutAdd()* provides a convenient way to register S functions to be called after a specified interval of time. If the function returns *T*, the task is rescheduled to run after the same interval. Alternatively, returning *F* discards the timer. One can programmatically remove the timer using *gSourceRemove()*, passing the value returned from *gTimeoutAdd()*.

By default, the function is called with no arguments. However, one can arrange to have it passed a value by specifying the object as the data argument in the call to *gTimeoutAdd()*. This is similar to the data argument for *gSignalConnect()*.

Idle tasks are run when there are no other events to process in the GLib event queue. These can be used to perform non-urgent background tasks. The interface is very similar to timeout functions. One registers an idle task with *gIdleAdd()* and this returns an identifier for the tasks. One can remove the task using *gSourceRemove()* (the same as with timers).

4.4   Field Accessors

Certain structures in the libraries bound by RGtk2 expose public fields. Generally, these fields are read-only. In fact, it is not possible to directly set a field with RGtk2 (this functionality, when allowed, is very rarely required). In order to access a field of a structure in RGtk2, use the following syntax:

```
obj[["fieldName"]]
```

4.5   Transparent Types

Conversion of primitive types between R and the libraries is fairly simple. A more complicated problem that RGtk2 attempts to solve is the conversion of simple, transparent C structures that are normally initialized manually and therefore lack a constructor. This problem could be solved in at least two ways. First, a function could be added that serves as a constructor for the structure. Unfortunately, this would break the strict adherence to the API, since a new function is introduced. Also, this solution violates the "spirit" of the API's design. The simple structures are meant to be initialized and manipulated without the extra baggage of function calls. Given these disadvantages, the alternative is favored: allowing the user to define an instance of such a type as an R list which is automatically converted to the corresponding C structure when passed to a wrapped function. When an instance of such a type is returned from a function, it is converted to its R list equivalent, preserving symmetry.

For example, suppose a user wished to construct an instance of GdkColor, a structure describing an RGB color with fields red, green, and blue. The following code would yield the color red: *c(65535, 0, 0)*. Here the fields for red, green, blue must be specified in the same order as they occur in the C structure definition. If the user desires an alternative order or does not wish to specify

8

all of the fields (they default to zero), then the list should be named according to the field names in the C structure. For example, red could be specified as *c(red=65535)*.

4.6   Special Constants

Special constants like the GDK keycodes and GTK stock id's are available in RGtk2. For the GDK keycodes, the C *GDK_ESCAPE* becomes *.gdkEscape* in R. For the stock id's, there is no difference between R and C, so *GTK_ CLOSE* is *GTK_ CLOSE*. The inconsistency is due to my uncertainty as to which form is better.

5   Advanced Concepts

To be documented.

5.1   RGtkDataFrame

5.2   Accelerated GtkListStore and GtkTreeStore loading

5.3   RGClosure

6   The Other Libraries

Most of the libraries besides GTK and the low-level GLib and GObject are concentrated on drawing. GDK provides basic drawing routines and access to low-level hardware devices. Cairo is a vector graphics library. Pango provides anti-aliased and internationalized fonts. GdkPixbuf is a specialized library for image manipulation. Of the other three libraries, Libglade is probably the most interesting. It allows the automatic construction of GTK GUI's from XML descriptions produced by the Glade tool. GtkMozEmbed allows the embedding of the Mozilla web component in a GTK container. The least interesting and likely least useful library is ATK, which adds accessibility device support to GTK.