R-Gtk Bindings

Duncan Temple Lang

October 31, 2002

Abstract

The Gnome desktop and related tools are becoming quite mature and are available on many machines and different platforms. Building GUIs within R is also becoming more common and important. While we can use Tk, bindings for Gtk would also be useful. They provide users with an alternative to Tk but more importantly

1. provide access to an actively developed toolkit with a large and increasing number of widgets;
2. allow direct access to and integration with Gtk-based applications from within R, allowing direct inclusion within GUIs rather than needing to create separate windows;
3. provide an easy environment for experimenting with and learning Gtk and the different libraries.
4. avoid Tcl!

We provide an automated mechanism for developing bindings for Gtk from R. The result is a "one-to-one" mapping of the Gtk API to R. Other packages can provide more R-like interfaces to unify and simplify the interface, and these will probably require more human intervention and design.

1

The Gnome desktop (window manager, office applications, etc.) and the underlying Gtk windowing toolkit are evolving into widely used and stable software that receive a great deal of attention. Gtk is available on both Windows and Mac OS X. Gtk provides a rich set of core and auxillary user-interface components (widgets). And, unlike other GUI toolkits, the development is continuing at a lively pace as evidenced by the recent release of Gtk 2.0.

Gtk underlies many of the Gnome applications, including

Gnumeric	A spreadsheet application.
AbiWord	A word-processor similar to Microsoft Word.
Gimp	An image manipulation program for creating and editing images.
Desktop	Numerous window managers and desktop components such as a task bar.
Tools	Many applications such as mail programs, running process viewers, etc.

2 The wxWindows Alternative

Why not? Simply because we don't have as much experience. In the future, we may use Slcc to generate bindings for it. This work is an initial effort in this respect of automated code generation for R to C.

3 Generating the Bindings

While we could develop an interface manually, it is more practical to write code that generates the interface. While this is more in direct and results in a less natural or "human" interface, the approach is easy to reproduce for new versions of Gtk and is vastly less error-prone and time consuming. Importantly, it will allow us to update the interface/bindings

for new releases of Gtk and the different toolkits. This is important given the speed at which Gnome development moves.

At present, I see three potential approaches to the automated construction of bindings. One is Gtk/Gnome-specific and uses a collection of class and method description files available as part of Gdk, Gtk, Gnomeui, etc. This is the approach used in generating most of the Gtk bindings. A more general approach to automated C binding generation is to use a mechanism that reads C code (or header files) and generates the code from these definitions. Both SWIG or our own Slcc package can be used in this approach.

3.1 Using the .defs files

The basic idea is that we have a complete description of the different libraries to which we wisht to interface. These come in **.defs** files that look something like

```
(define-object GtkTreeItem (GtkItem)
  (fields (GtkWidget subtree)))

(define-func gtk_tree_item_new
  GtkWidget
  ())

(define-func gtk_tree_item_new_with_label
  GtkWidget
  ((string label)))
```

Figure 1: GtkTree

Figure 2 gives the definitions related to the *Button* class. The primary question is what should the the R interface to this class contain? Specifically, what is the minimal interface that provides complete acess to the class and on which we can build an R interface that is consistent across the different Gtk classes.

We can create the class hierarchy specification in C or R. We can also maintain it separately in R so that it can provide reflectance information.

For each of the widget constructors, we can add a show argument which controls whether the widget is automatically shown. This can default to TRUE if the object is not a container that can have other widgets added to it.

Each of the 5 "methods" are quite simple. They take the button object as their only argument and have no return value.

More interesting *Layout Table*, *HBox*

Optional arguments, i.e. null-ok.

```
gtkLayout <-
function(hadjustment = NULL, vadjustment = NULL)
{
  gtkInheritsCheck(hadjustment, vadjustment, "GtkAdjustment")
  # or two separate calls.
  # gtkInheritsCheck(vadjustment, "GtkAdjustment")

  w <- .Call("S_gtk_layout_new", hadjustment, vadjustment)
  class(w) <- gtkClass("GtkLayout")
}
```

Again, all the methods are relatively simple mappings.

```
gtkLayoutMove <-
```

```
(define-object GtkButton (GtkContainer) (fields (bool in_button) (bool
button_down)))

(define-func gtk_button_get_type GtkType ())

(define-func gtk_button_new GtkButton ())

(define-func gtk_button_new_with_label GtkButton ((string label)))

(define-func gtk_button_pressed none ((GtkButton button)))

(define-func gtk_button_released none ((GtkButton button)))

(define-func gtk_button_clicked none ((GtkButton button)))

(define-func gtk_button_enter none ((GtkButton button)))

(define-func gtk_button_leave none ((GtkButton button)))
```

Figure 2: GtkButton

```
gtkButton <-
function(label = NULL, show = T)
{
 if(is.null(label))
   w <- .Call("S_gtk_button_new")
 else
   w <- .Call("S_gtk_button_new_with_label", as.character(label))

 class(w) <- c("GtkButton", "GtkContainer", "GtkWidget", "GtkObject")

 if(show)
   gtkShow(w)

 w
}
```

Figure 3: *Button*

```
pressed.GtkButton <-
function(w)
{
  gtkCheckInherits(w, "GtkButton")
  .Call("S_gtk_button_pressed", w)
}
```

Figure 4: *Button* pressed method

```
(define-object GtkLayout (GtkContainer))

(define-func gtk_layout_new
  GtkWidget
  ((GtkAdjustment hadjustment (null-ok) (= "NULL"))
   (GtkAdjustment vadjustment (null-ok) (= "NULL"))))

(define-func gtk_layout_put
  none
  ((GtkLayout layout)
   (GtkWidget child)
   (int x) (int y)))

(define-func gtk_layout_move
  none
  ((GtkLayout layout)
   (GtkWidget child)
   (int x) (int y)))

(define-func gtk_layout_set_size
  none
  ((GtkLayout layout)
   (uint width) (uint height)))

(define-func gtk_layout_freeze
  none
  ((GtkLayout layout)))

(define-func gtk_layout_thaw
  none
  ((GtkLayout layout)))

(define-func gtk_layout_get_hadjustment
  GtkAdjustment
  ((GtkLayout layout)))

(define-func gtk_layout_get_vadjustment
  GtkAdjustment
  ((GtkLayout layout)))

(define-func gtk_layout_set_hadjustment
  none
  ((GtkLayout layout)
   (GtkAdjustment adjustment)))

(define-func gtk_layout_set_vadjustment
  none
  ((GtkLayout layout)
   (GtkAdjustment adjustment)))
```

Figure 5: Layout

```
function(w, child, x, y)
{
  gtkCheckInherits(w, "GtkLayout")
  gtkCheckInherits(child, "GtkWidget")
  .Call("S_gtk_layout_move", w, child, as.integer(x), as.integer(y))
}
```

We might want to have S-like arguments which would allow x to be specified as an integer vector of length 2 to provide both x and y values. This can be added by an additional library.

Converting to uints (in the `gtk_layout_set_size()` method) can be done by enforcing non-negativity. We want a general set of converters here.

3.2 Table

Enums. See GtkAttachOptions in `gtk_table_attach()`. Optional arguments also in `gtk_table_attach()`

3.3 Notebook

Fields. Enum GtkPositionType. Want accessors for the fields: set and get methods. Can we do a set? What if there are set methods in the API routines. Look for a `gtk_notebook_set_<field_name>()`

Enum GtkPackType in `gtk_notebook_set_tab_label_spacing()`

3.4 Checking Gtk Object Types

The check for a valid Gtk object type is given by.

```
gtkInheritsCheck <-
#
# if critical is TRUE, an error is generated
# in the case that w does not inherit from the
# specified class.
# If it is FALSE, a warning is generated.
# If critical is a string (character vector of length 1)
# it is passed directly to stop() and used as the error message.
# This allows the caller to give more context-specific
# messages.
function(w, klass,  critical = TRUE)
{
 if(!inherits(w, klass)) {
   if(is.character(critical))
     stop(critical)
   else if(is.logical(critical) && critical)
     stop(paste("object ", w, "of class", class(w), "isn't a",klass))
 }

 return(TRUE)
}
```

We only have single inheritance.

The key step in the generation of the code is to process the defs files. Fortunately, others have done this already. We can leverage the code used in generating bindings for other languages. Specifically, we will use the Python code that is used in the pygtk bindings. Others are also available to us, such as the Java and Perl mechanisms for generating bindings in those languages. We will focus on the Python mechanism. Note that there exist packages (from Omegahat) to invoke from R functions and methods in each of these languages and we will exploit

```
(define-object GtkTable (GtkContainer))

(define-func gtk_table_new
  GtkWidget
  ((int rows)
   (int columns)
   (bool homogenous)))

(define-func gtk_table_attach
  none
  ((GtkTable table)
   (GtkWidget child)
   (int left_attach)
   (int right_attach)
   (int top_attach)
   (int bottom_attach)
   (GtkAttachOptions xoptions (= "GTK_EXPAND|GTK_FILL"))
   (GtkAttachOptions yoptions (= "GTK_EXPAND|GTK_FILL"))
   (int xpadding (= "0"))
   (int ypadding (= "0"))))

(define-func gtk_table_attach_defaults
  none
  ((GtkTable table)
   (GtkWidget child)
   (int left_attach)
   (int right_attach)
   (int top_attach)
   (int bottom_attach)))

(define-func gtk_table_set_row_spacing
  none
  ((GtkTable table)
   (int row)
   (int spacing)))

(define-func gtk_table_set_col_spacing
  none
  ((GtkTable table)
   (int column)
   (int spacing)))

(define-func gtk_table_set_row_spacings
  none
  ((GtkTable table)
   (int spacing)))

(define-func gtk_table_set_col_spacings
  none
  ((GtkTable table)
   (int spacing)))

(define-func gtk_table_set_homogeneous
  none
  ((GtkTable table)
   (bool homogeneous)))

(define-func gtk_table_resize
  none
```

3.5 PyGTK

The PyGTK distribution provides Python code for parsing the defs files and generating the C and Python code to interface to the code represented by those defs files. Basically, this runs through a defs file and calls methods for defining objects, functions, enumerations, etc. We can extend this primary class that does the parsing and bypass the code generation. Let's call our class **SDefsParser**. We can arrange for this Python class to call R functions that accumulate the information about these different types. Then, when we have processed all of the defs files, the complete information about the code described in the defs files will be in R and we can process it using standard tools.

We want to define a Python class **SDefsParser** with essentially five methods and a constructor function. The methods are

define˙func()
define˙object()
define˙boxed()
define˙enum()
define˙flags()

3.5.1 Objects

This is called when the contents of a define-object element in a defs file is completed. We are given the name of the class being defined, the name of the parent class and a list of any fields defined for that class.

3.5.2 Functions

This is called each time a define-func element is completed when parsing a defs file. The arguments are the name of the function, the name of the return type and a list of the arguments for the function. Each element of the argument list is a pair giving the name of the type of the argument and the name for the argument.

3.5.3 enumerations

We use this to collect definitions of enumerations, i.e. symbolic constant groups. We are given the name of the enumeration (e.g. GdkInputMode) and a tuple of name-value pairs. These are all symbolic, but we can infer the actual values since they start at 0. (Check this!) We don't want users to specify values by number anyway.

3.5.4 flags

Not certain what the difference are between a flag and an enumeration.

3.5.5 Boxed

Not quite certain yet. These are opaque data types that are to be treated as references to foreign objects.

3.6 SDefsParser

Given the basic mechanism from the **FilteringParser**, we can extend the class in at least two different ways. We can use Python to gather the information and query it from R at the end of the parsing. Alternatively, we can have Python inform R as each entity is processed.

3.6.1 Collection in Python

Perhaps the simplest and most practical approach is to create and start the parser from R and cumulate the different definitions in the Python code. When the entire parsing is complete, then we can query transfer the results from Python to R by accessing the fields of the parser in which the results have been stored. The Python code for this is straightforward, merely storing functions, objects, etc in dictionaries/hash tables. You can find it in `Sgenerate.py`.

3.6.2 Notifying R for Each Entity

The definition of our **SDefsParser** is quite simple. Each method is simply a call to an R function passing the method's arguments to that function. When we create an instance of this class, we pass it a list of five functions corresponding to the different methods.

```
import generate

class SDefsParser(generate.FilteringParser):
    """An interactive parser for examining the output from the real parser"""
    def __init__(self, input, handlers):
        generate.FilteringParser.__init__(self, input)
        self.handlers = handlers

    def define_object(self, name, parent=(), fields=()):
        R.call(handlers[0], name, parent, fields)

    def define_func(self, name, retType, args):
        R.call(handlers[1], name, retType, args)

    def define_enum(self, name, *values):
        R.call(handlers[2], name, retType, args)

    def define_flags(self, name, *values):
        R.call(handlers[3], name, retType, args)

    def define_boxed(self, name, reffunc = None, unreffunc=None, size=None):
        R.call(handlers[4], name, reffunc, unreffunc, size)
```

From within R, we use a closure to define the collection of five functions.

```
SDefsHandlers <-
function()
{
 classes <- list()
 functions <- list()
 enums <- list()
 boxes <- list()

 object <- function(name, parent, fields) {

 }

 func <- function(name, retType, args) {
   functions[[name]] <- list(retType, args)
 }

 return(list(object=object, func=func, enum=enum, flags=flags, boxed=boxed))
}
```

3.7 The Java-Gnome Binding Generator

3.8 Perl-GTK

3.9 Slcc

3.10 SWIG