RGtk2 - A GUI Toolkit for R

August 26, 2005

1 Introduction

1.1 Motivation

RGtk2 enables the R programmer to construct graphical user interfaces with GTK+, a GUI toolkit that is very commonly used by linux desktop applications and so is popular with the open-source community. The R platform greatly benefits from access to GTK+ in that it allows novice users, such as biologists, to capitalize on the analytical functionality of R, without the hindrance of the learning curve associated with a console-driven interface. For example, a graphical interface could guide a biologist through a microarray data analysis task driven by Bioconductor. GTK+ is a vast improvement over the existing GUI toolkit for R, tcl/tk, as GTK+ is more advanced and, by virtue of its popularity in open-source applications, is capable of integrating interface functionality from a wide-range of other projects, including GGobi and Mozilla Firefox.

1.2 Background

The original RGtk, based on the now obsolete GTK+ version 1.2, was developed by Duncan Temple Lang. About 4 years ago, GTK+ 1.2 was overhauled and renamed to GTK2. The fundamental GTK object system was abstracted into a separate library called GObject, part of GLib. GTK2 also takes advantage of the new font rendering library, Pango. Many widgets were added, removed, and heavily altered. GTK2 has more sophisticated widgets, prettier text, and a more elegant foundation than its predecessor. RGtk2 is an attempt to catch up with the evolution of GTK+. The goal is virtually complete support for the latest version of GTK2 (2.8.0) and its underlying libraries.

1.3 Scope

GTK2 is dependent upon a collection of libraries, all of which RGtk2 aims to bind to R.

GLib: Handles common tasks such as string manipulation, linked lists, event looping, etc.

GObject: Contains object and dynamic type system, including properties and signals.

ATK: Defines a common interface for accessibility technologies, implemented by GTK.

Cairo: Vector graphics library with which GTK+ widgets are drawn.

Pango: Renders text, with full support for internationalization.

GDK: Handles interaction with native window system, including drawing and events.

GdkPixbuf: Renders pixbufs, integrated with GDK.

GTK: Provides the widgets for the GUI.

Except for the first two, GLib and GObject, all of these libraries are fully bound to R by RGtk2. GLib and GObject are partially bound to the extent necessary for support of the others. A library binding consists of several components:

Functions: All functions are wrapped with automatic type conversion of the parameters and return value(s).

Fields: The values for fields of non-opaque structures are retrieved and converted. This is read-only access.

Callbacks: User R functions are wrapped as typed C callback functions.

Converters: Library-specific types may require special conversion.

2 Design

2.1 Goals

The design goals of the project are two-fold. First, the bindings must be complete and consistent with the bound API. This simplifies documentation in that there is a more-or-less one-to-one correspondence between RGtk functions and the API functions. This also ensures that the R programmer has complete control over the API without any gaps or deficiencies in functionality. Whatever the C programmer can do, the R programmer should be able to do. Second, interaction with RGtk must be simple and familiar to the R programmer. Foreign C concepts such as memory management, return-by-reference parameters, and type casting must be hidden or adapted to their R equivalent. The user should be able to enjoy the benefits of GTK+ without knowing that it is implemented in a foreign language.

2.2 Central Problem

Given the broad scope of the project, it is obvious that manual implementation of the bindings would be extremely tedious and time consuming. In order to avoid this, much of the code was autogenerated. Autogeneration also enhances the maintainability of the project, since improved code can be uniformly and automatically generated across all cases.

GTK+ and other GObject-based API's are defined according to a scheme-based format called *defs*. The definitions describe an API's object hierarchy, function names, parameter types, structure fields, etc. They are annotated with information that assists in more subtle aspects, such as memory management. The most mature GTK+ language binding, pygtk, maintains a set of reference *defs* files and provides python scripts for their generation and parsing. RGtk employs these scripts for parsing the *defs* files via RSPython. The result is converted to R and C binding code through the use of a code generation library implemented in R. The *defs* format was overhauled along with GTK+ in the transition to GTK2, and it contains much more information than was provided to the original RGtk. However, there are still a small number of functions that require manual implementation, usually in the case of complicated memory management.

2.3 *Defs* Shortcomings

The annotated definitions are very helpful in generating code. Unfortunately, the pygtk authors manually implement a large portion of their bindings, so they rely less on the *defs* information. The pygtk definitions, therefore, are buggy and do not contain all of the information necessary to come close to completely generating all of the bindings automatically.

The first task of building RGtk was the cleaning and annotating of the *defs* files. An example of an annotation is the specification of a parameter as in, out, or in-out. This allows the code generator to know when to accept a parameter as input to a function and when to only return it as part of the result. Other additions include defining explicit callback types for generating wrappers and specifying the types of linked list elements, which is useful for type conversion. Some of these modifications were not explicitly supported by the *defs* specification, but nor were they disallowed. The pygtk parsers were extended to handle the new features, since they were written only to handle the subset of the specification employed by the pygtk definitions.

2.4 Type Conversion

Every component of the bindings requires type conversion to some extent. The code generator attempts to write code that converts C types to and from R. Primitive types, such as double [numeric], int [integer], and char* [character], are perhaps the simplest to convert. Opaque structures such as GObjects and "boxed" types are passed to R as externalptr's, with the class attribute set to

a character vector representing the type hierarchy of the object. Collections of these types, in the form of arrays and linked lists, are simply converted by iterating over the data structures. The original version of RGtk supported this functionality, except for linked lists and some array types.

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 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)*.

2.5 Memory Management

Memory management in GObject-based libraries is based on the data type. GObjects are reference counted, while boxed types are explicitly freed after use. The memory persistence of other structures is either freed on demand, based on reference counting, or is handled internally. Finalizer functions for the boxed types are specified in the *defs*. The code generator registers these as the finalizers for the corresponding externalptr in R. The reference counting of objects is also handled automatically. Other structures may be special-cased in the generator or dealt with manually.

All of these mechanisms are dependent on whether RGtk "owns" the memory of a returned value, which is also specified in the *defs* files. For example, if RGtk owns an object's memory, it does not need to increase the reference count. If RGtk does not own an instance of a boxed type, then it should not register its externalptr for finalization. One shortcoming of the *defs* format is that it is not possible to specify the ownership of memory returned by reference, so these cases must be dealt with using heuristics and manual implementation.

2.6 Adapting to R

Memory management is just one of the annoyances of C that R programmers are happy to avoid. One example is the need to specify the lengths of arrays (including strings) when passing them to functions, unless it is assumed that the arrays are NULL-terminated. The code generator uses heuristics to identify these parameters and does not require R to provide them. The wrapping code determines the length of arrays automatically. Another complication is the ability of C to return values by reference. These parameters, in addition to the return value, are returned to R compiled as a list. This avoids trying to emulate the foreign concept of return-by-reference in R. As a final example, certain errors that occur in GLib-based libraries are described by a returned GError structure. In R, libraries often alert the user to a problem via a printed warning. The RGtk2 user may specify whether to print such a warning when a GError is returned by passing a parameter to the wrapped function. If printing is not requested, the user can still inspect the list structure containing the fields of the GError.

3 Other Issues

3.1 The Event Loop

Many GTK2 widgets have complex behavior that requires the execution of timeout and idle tasks. In order to reliably invoke these tasks, the GTK event loop must be the primary R event loop. The strategy to achieve this depends on the platform. On Linux, the REventLoop package by Duncan Temple Lang provides a framework for replacing the default Read-Eval-Print Loop with a foreign loop, without losing control of the console. RGtk2 provides a GTK2 (actually GLib 2.x) implementation for the REventLoop package. On Windows, the solution is somewhat simpler. The tcl/tk package invokes a function named *tcl_do* when it is waiting for console input, so that it can still respond to tcl/tk events. RGtk simply redefines *tcl_do* so that it checks for GTK events instead. Both of these solutions are satisfactory, but they are kind of ugly hacks.

3.2 Compatibility

The GTK+ API is in constant flux; it changes with each minor version. RGtk must somehow accomodate the many different versions without overly complicating the code generation process. For example, one solution would be to autogenerate code with preprocessor directives allowing conditional compilation based on the user's GTK version. However, this would greatly complicate the process. It also would not account for the R side of the wrappers. Instead, the RGtk code was simply branched, so that as of this writing three minor versions of GTK are supported (2.8.0, 2.6.0, and 2.4.0). As GTK and RGtk advance, the previously branched versions will continue to be maintained by merging version independent code enhancements between them.

4 Additional API Support

4.1 iWidgets

Simon Urbanek's iWidgets is an attempt to establish a simple API for quickly constructing GUI's in R. It is very minimalistic and thus it should be relatively straightforward to provide iWidget implementations based on any mature widget toolkit. In theory, it should allow the R programmer to easily create a native GUI without concern for any quirks or minutia associated with a particular platform or toolkit. The programmer should be able to focus on the statistics without getting bogged down in GUI building. RGtk2 provides a GTK2 implementation of iWidgets, which is perhaps as close as one can get to native support on Linux.

4.2 Extra Libraries

Two additional GTK-related libraries are included with RGtk2. The first is libglade, which allows one to create a GUI by reading an XML specification at runtime. The Glade graphical GUI builder exports this XML, so even someone with little GUI programming experience can quickly and easily construct a complex interface. Also, RGtk2 includes GtkMozEmbed bindings for embedding a Mozilla Firefox renderer into an RGtk2 application. This demonstrates the usefulness of binding to a toolkit that is used by many major open source applications.

5 Conclusion

In short, RGtk2 has achieved its goals of being complete and consistent without sacrificing simplicity and familiarity to the R programmer. A total of eight libraries are completely (and two more partially) bound to R. Every attempt is made to keep a one-to-one correspondence with the C API. Technologies such as glade and iWidgets greatly simplify the task of creating GUI's using RGtk2. Finally, foreign C concepts like memory management are avoided and assimilated, ensuring that the learning curve for an R developer is as shallow as possible.