and update the data. The **AnnBuilder** approach addresses issues 2, 3 and 5. Data reduction is under the control of the data assembler. The notion of a package that generates packages is our method of dealing specifically with issues 2 and 3. The repository mechanism allows the data assembler to distribute the assembled data (they may choose to restrict distribution only to specific sites).

Acknowledgment

Robert Gentleman's work is supported in part by NIH/NCI Grant 2P30 CA06516-38.

Bibliography

Affymetrix. *Affymetrix Microarray Suite User Guide*. Affymetrix, Santa Clara, CA, version 5 edition, 2001. 22

A Brazma, P Hingamp, and J Quackenbush. Minimum information about a microarray experiment: towards standards for microarray data. *Nature Genetics*, 29:365–371, 2001. 23

Jianhua Zhang, Vincent Carey, and Robert Gentleman. An extensible application for assembling annotation for genomic data. *Bioinformatics*, 19, 2002. 24

Jianhua Zhang, Vincent Carey, A. J. Rossini, and R. C. Gentleman. Annbuilder - an open source application for genomic data annotation. *JSS*, in preparation. 24

Robert Gentleman
DFCI
rgentlem@jimmy.harvard.edu

Changes to the R-Tcl/Tk package

by Peter Dalgaard

Introduction

In a previous issue of R News, I gave a small tutorial on the basic use of the `tcltk` package for creating graphical user interfaces (Dalgaard, 2001). Since then, there has been a number of changes to the package, and the purpose of this paper is to outline the new possibilities offered by the modified interface.

Some of the changes were incompatible with old code. In particular, some of the examples in Dalgaard (2001) no longer run. The last section of the paper contains a revised version of the scripting widget.

Control variable changes

The old-style interface to Tcl variables allowed you to change the Tcl variable `foo` using syntax like

```
tclvar$foo <- "Hello, World"
```

With this approach, `foo` becomes a global Tcl variable, and there is no way to ensure that two R functions do not accidentally use the same variable name. (This is less of a problem in Tcl because there you can use techniques like prefixing the variable with the window name.)

Therefore, a new technique is introduced using the object class `"tclVar"`. The creator function for that class is `tclVar()` and it works by creating a new Tcl variable. You generally do not need to care about the names on the Tcl side, but they are simply `::RTcl1`, `::RTcl2`, and so forth (the `::` prefix ensures that they are in Tcl's global namespace). The accessor function is `tclvalue()` and a typical usage is like this:

```
foo <- tclVar()
tclvalue(foo) <- "Hello, World"
```

Notice that you have to create the variable before it can be used.

The `"tclVar"` objects are subject to finalization, so that when they go out of scope, the garbage collector will eventually remove their Tcl counterparts.

Tcl objects

The C interface to Tcl contains the notion of *dual ported* objects. All Tcl objects have a *string representation*, but they can also have a "dual personality" as doubles, integers, or lists of strings, doubles, or integers. (There are other possibilities but we ignore them here.)

This allows you to access Tcl variables without going via the string representation. The latter can be a major pain because of "quoting hell": The need to escape certain characters because they otherwise have special meaning to the Tcl interpreter.

The interface from R maps double, integer, and character vectors to Tcl lists of the corresponding object types. This works via an accesor function called `tclObj()`. An example should convey the gist of the interface:

```
> foo <- tclVar()
> tclObj(foo) <- c(pi,exp(1))
> tclvalue(foo)
[1] "3.14159265359 2.71828182846"
> as.character(tclObj(foo))
[1] "3.14159265359" "2.71828182846"
> as.double(tclObj(foo))
[1] 3.141593 2.718282
> as.integer(tclObj(foo))
[1] NA NA
```

Notice that the elements can not be interpreted as integers, and that `as.integer` in that case returns `NA` rather than attempting to truncate the values.

The return value from `tclObj()` is an object of class `"tclObj"`. It is only possible to modify such objects when there is an underlying Tcl variable. Also, the only way to modify a Tcl object is to extract it in its entirety, modify the R representation, and store it back into the Tcl variable.

The issue that prompted the development of this interface was the handling of listboxes. Using Tcl objects, this can be done as simply as

```
mylist <- tclVar()
tkpack(lb <- tklistbox(tt <- tktoplevel(),
                       listvariable = mylist))
tclObj(mylist) <- month.name
```

Using the string representation, you would need something like

```
tclvalue(mylist) <-
    paste(month.name, collapse=" ")
```

which may not look too bad, but if you need to have a list elements with spaces inside things get complicated.

With the Tcl object interface, you have a direct connection to the list contents, and can do things like

```
as.character(tclObj(mylist))[
    as.integer(tkcurselection(lb))+1]
```

which will return the text of any selected items. Notice that it is necessary to add 1 since Tcl uses zero-based indexing.

Return values

The return values from `.Tcl` calls (and thus most of the interface functions, the main exception being the widget creation commands) are now objects of class `"tclObj"` instead of a string value. This is useful for much the same reasons as described in the previous section.

For instance, if you turn on multiple selection mode for a listbox with

```
tkconfigure(lb, selectmode="multiple")
```

then the string representation of

```
tkcurselection(lb)
```

might be `"0 3 4 5"` and it would be the programmer's responsibility to parse the string into four integers. With the object representation, you push that responsibility into the Tcl layer and it becomes possible to access the vector of indices directly using `as.integer(tkcurselection(lb))`.

Changing the return value created some incompatibilities with some earlier code. Fortunately the extent of the problems is limited since return values are often ignored, and you can alway restore compatibility by taking `tclvalue` of the returned object.

Callback changes

Several new formats have been introduced for specifying callbacks. Some of these just offer notational convenience, but some were necessary because the old interface had no way to invoke Tcl's `break` command. For instance you bind an action to, say, `<Control-Return>` then any other bindings for `<Return>` will also execute, unless `break` is used. This can now be coded as

```
tkpack(txt <- tktext(tktoplevel()))
tkbind(txt, "<Control-Return>",
       expression(print(tkget(txt,"0.0","end")),
                  break))
```

Compared to the old style where a callback had to be a function, the new rules allow some simplification when there are no arguments. E.g.,

```
but <- tkbutton(tt, text="Hit me!",
                command=expression(cat("OW!!")))
```

Conversely, if you do need to be able to pass arguments to the callback *and* need to use `break`, then you can do so by specifying a function for one or more elements of the callback expression. That is, you could use

```
tkbind(txt, "<Control-Button-1>",
       expression(function(x,y) cat(x,y,"\n"),
                  break))
```

to create a callback that prints out the mouse cursor position, without performing the text cursor motion bound to `<Button-1>`.

New version of the script widget

The script-window application in Dalgaard (2001) got hit rather badly by the interface changes. Below is a version that works with the new interface.

Notice that it is necessary to insert `tclvalue()` constructions in several places, even when the return values are only used as arguments to Tcl/Tk routines. You can sometimes avoid this because the default treatment of arguments (in `.Tcl.args()`) is to preprocess them with `as.character()`, but for objects of class `"tclObj"` this only works if there are

no whitespace characters in the string representation. The contents of the script window and the files that are read can obviously contain spaces and it is also not safe to assume that file names and directory names are single words.

Notice also that several new functions have been added to the interface, `tkopen`, `tkclose`, `tkfile.tail`, etc. (Beware that `tkread` was not added until R version 1.6.1; for version 1.6.0 you need `tkcmd("read",...)` instead.)

```
tkscript <- function() {
  wfile <- ""
  tt <- tktoplevel()
  txt <- tktext(tt, height=10)
  tkpack(txt)
  save <- function() {
    file <- tclvalue(tkgetSaveFile(
      initialfile=tclvalue(tkfile.tail(wfile)),
      initialdir=tclvalue(tkfile.dir(wfile))))
    if (!length(file)) return()
    chn <- tkopen(file, "w")
    tkputs(chn, tclvalue(tkget(txt,"0.0","end")))
    tkclose(chn)
    wfile <<- file
  }
  load <- function() {
    file <- tclvalue(tkgetOpenFile())
    if (!length(file)) return()
    chn <- tkopen(file, "r")
    tkinsert(txt, "0.0", tclvalue(tkread(chn)))
    tkclose(chn)
    wfile <<- file
  }
  run <- function() {
    code <- tclvalue(tkget(txt,"0.0","end"))
    e <- try(parse(text=code))
    if (inherits(e, "try-error")) {
      tkmessageBox(message="Syntax error",
                   icon="error")
      return()
    }
    cat("Executing from script window:",
        "-----", code, "result:",  sep="\n")
    print(eval(e))
  }
  topMenu <- tkmenu(tt)
  tkconfigure(tt, menu=topMenu)
  fileMenu <- tkmenu(topMenu, tearoff=FALSE)
  tkadd(fileMenu, "command", label="Load",
        command=load)
  tkadd(fileMenu, "command", label="Save",
        command=save)
  tkadd(topMenu, "cascade", label="File",
        menu=fileMenu)
  tkadd(topMenu, "command", label="Run",
        command=run)
}
```

Future developments

The **tcltk** package is still nowhere near its final state, and there are several changes and enhancements in the works.

The use of Tcl objects has proven to be a big advantage, and I'd like to extend its use so that we as far as possible can avoid direct use of string representations. In particular, I believe that it should be possible to modify the current `.Tcl` interface to use Tcl objects and thereby avoid the "quoting hell" associated with passing arbitrary character strings. Also, it ought to be possible to pass Tcl objects directly to Tcl commands, which would remove the need for many calls to `tclvalue`.

We need to become able to take advantage of the various extensions that exist to Tcl and Tk. Sometimes this is almost trivial, but in other cases attempting to do so reveals shortcomings of the current interface. For instance, it is tempting to use the features of the TkTable widget to implement a better spreadsheet-like data editor than the raw X11 one we currently have. However, this entails finding a way to return a value from a callback, and/or an extension of the interface to Tcl variables that can deal with arrays. Notice that it is not quite trivial to return values from a callback, since we cannot in general expect to convert an arbitrary R object to something that makes sense in Tcl.

Another thing that I have been working on is the addition of a Tk-based console for R. The main reason is that you cannot add menus to a terminal window, and a free-floating toolbar would probably not be too user-friendly. R-1.6.0 and later (Unix versions) have a couple of stubs to allow redirection of input and output to Tcl functions, and I have an as yet unpublishable sketch of the rest.

Finally, there is the issue of writing a graphics driver. Luke Tierney has a fairly decent solution in the **tkrplot** package, but it would be interesting to incorporate more features of the Tk canvas, which does allow some manipulations that you don't have with standard devices. E.g., you can support simple interactive graphics by tagging plot elements, so that you could move or delete them afterwards. Unfortunately, the Tk canvas is limited with respect to clipping and rotation of text, but the Zinc extension (http://www.openatc.org/zinc) looks promising.

Bibliography

Peter Dalgaard. A primer on the R-Tcl/Tk package. *R News*, 1(3):27–31, September 2001. URL http://CRAN.R-project.org/doc/Rnews/. 25, 26

Peter Dalgaard
Department of Biostatistics
University of Copenhagen, Denmark
p.dalgaard@biostat.ku.dk