A Structure for Interfaces from R

John M. Chambers

July 1, 2016

(*Extending R*, Chapter 13)

Language Interfaces
Design goals

convenience:

- Programming an application package to use an interface should be straightforward.
- The *users* of the application should be largely unaware of the interface, just doing ordinary R computations.

generality:

- Any server language function or class of objects should be available.
- Results should be available as the appropriate R class of object.

consistency: The interface programming for similar computations should be independent of the details of the server language implementation.

Preliminaries

- The languages here are those that evaluate user calls to functions or methods *and* have some form of class structure. (Python, Java, Julia, Perl, JavaScript, ...).
- I'll call these *server languages*, but that doesn't imply a particular communication mechanism.
- We're talking about interfaces *from* R; interfaces *to* R exist also, but our topic is extending R.
- All the packages mentioned are on github at `github.com/johnmchambers`
- This is new stuff; you're welcome to try it out but there isn't a lot of experience yet.

The XR Interface Structure

For extending R using interface(s), think of three levels of packages:

1. The XR package defines
   - a class of *evaluators* that communicate to a server language;
   - methods for creating *proxy* objects, functions and classes;
   - functions for *data conversion* between the languages.
2. An interface package for individual languages (XRPython, XRJulia) to create proxies and do anything else needed.
   The interface for the language specializes the XR structure:
3. Application packages (what you would write): they will create functions and classes proxies for the server language(s), using the interface for one or more languages, usually through simple function calls (e.g., `JuliaFunction()`, `setPythonClass()`.

Evaluators

An evaluator is an object from some interface class (e.g., `"PythonInterface"`) that extends the `"Interface"` class in the XR package (a reference class).

- Evaluators have methods to evaluate expressions, get and send objects, carry out commands in the server language.
- The methods look the same and work the same for all languages, except when it makes no sense (e.g., Java has methods, not functions).
- The structure is specialized to a server language by overriding low-level methods for evaluators and by OOP methods in R and often in the server language as well.

Evaluators

An evaluator is an object from some interface class (e.g., `"PythonInterface"`) that extends the `"Interface"` class in the XR package (a reference class).

- XR manages all the evaluators. In particular, if there is a "current" evaluator for a language, that can be used automatically.
- The result of computing any expression is returned to R. That object can be used in any subsequent evaluator method.
- Everything is based on the layer of evaluators & methods, but proxy functions, proxy classes and functional shortcuts hide evaluators from most applications.

Evaluators

```
ev <- RPython()

ev$Import("xml.etree.ElementTree")

hamlet <- ev$Call("xml.etree.ElementTree.parse",
                                        "hamlet.xml")

ev$Eval("%s.findtext('TITLE')", hamlet)
```

Proxy Objects

XR interfaces call arbitrary functions by assigning the value of the call in the server and returning from R a *proxy* for that object.

- Current interfaces convert & return scalars; anything else is a proxy.
- The evaluator assigns the value of an expression and returns a proxy containing the name used. So supplying the proxy later on just accesses the object by name.
- Note that the *evaluator* does assignments in the server; *you* don't need to and usually shouldn't.
- If you need the result to be converted, there are methods and optional arguments to force that.

Proxy Functions

Often, you want to call a function in the server language, from R. Functions in R that are *proxies* for the server language functions make this simple, and eliminate the need to use an evaluator explicitly.

- A one-line expression in the application package creates the proxy function.
- These are from a subclass of R functions, so users call them just like any function.
- If server languages have metadata about functions, that may be included in the proxy (e.g., Python functions can have documentation.)
- Application packages are encouraged to include their own server language software, and make proxy functions for that.

Proxy Functions

```
parseXML <- PythonFunction("parse", "xml.etree.ElementTree")

hamlet <- parseXML("hamlet.xml")

getSpeeches <- PythonFunction("getSpeeches", "thePlay")

hSpeeches <- getSpeeches(hamlet)
```

Proxy Classes

If a proxy object comes from a particular class in the server language, defining a corresponding *proxy class* in R allows fields and methods to be used directly from R.

- A one-line expression in the application package creates a (reference) class in R that is a proxy for the specified server language class.
- Metadata about the server language class is used to define the fields and methods for the proxy.
- Server language fields and methods can be used like fields and methods in R; for example, `x$title`, `x$append()`.

Proxy Classes

- If the proxy class is defined, proxy objects from that class are promoted to the class automatically.
- As with functions, it's often valuable for the application package to define its own server language classes and make proxies for them.

```
setPythonClass("Speech", module = "thePlay")

last <- hSpeeches$pop()
last$speaker
```

Specializing to the Server Language

`$ServerExpression(expr,...)` Encode expr, objects as string

`$ServerEval(string, key, .get)` Evaluate string, return proxy or convert.

`$ServerFunctionDef(what,...)`

`$ServerClassDef(ClassName,...)`

`$ServerSerialize(key, file)`

`$ServerUnSerialize(file)`

`$ServerAddToPath(directory, pos)`

`$Import(module,...)`

`$Source(file)`

Data Conversion

The XR approach to converting objects between R and the server language is to provide conventions for explicit representation of arbitrary objects, plus mechanisms for this to be specialized.

- Basic objects in R (vectors and some other types) are converted using the JSON representation: scalars, lists, dictionaries.
- Conventions using specially named elements in dictionaries allow objects to be converted recursively in terms of their fields.
- The general mechanism is specialized by defining methods for two generic functions in R: `asServerObject()` and `asRObject()`.
  Classes in the two languages that match (e.g., arrays in Julia and R) handled by OOP methods in one or both languages.