R and the World:
Interfaces between Languages

John M. Chambers

May 19, 2015
R and Interfaces

- Bell Labs’ S, the predecessor of R, was designed (ca. 1976) as an interactive language interfacing to Fortran libraries.
Bell Labs’ S, the predecessor of R, was designed (ca. 1976) as an interactive language interfacing to Fortran libraries.

Interfaces to a growing diversity of languages (e.g., the DBI interfaces to SQL-based DBMS) are a key asset in R.
R and Interfaces

- Bell Labs’ S, the predecessor of R, was designed (ca. 1976) as an interactive language interfacing to Fortran libraries.
- Interfaces to a growing diversity of languages (e.g., the DBI interfaces to SQL-based DBMS) are a key asset in R.
- A goal for some recent work is a uniform structure for interfaces that improves their ease of use and generality.
Understanding R

Three Principles

**OBJECT**

Everything that exists in R is an object.

**FUNCTION**

Everything that happens in R is a function call.

**INTERFACE**

We want to use the best software. If it's not in R let's try to use it through an effective interface.
Understanding R

Three Principles

OBJECT

Everything that exists in R is an object.

FUNCTION

INTERFACE
Understanding R

Three Principles

OBJECT

Everything that exists in R is an object.

FUNCTION

Everything that happens in R is a function call.

INTERFACE

We want to use the best software. If it's not in R let's try to use it through an effective interface.
Understanding R

Three Principles

**OBJECT**

Everything that exists in R is an object.

**FUNCTION**

Everything that happens in R is a function call.

**INTERFACE**

We want to use the best software. If it’s not in R let’s try to use it through an effective interface.
Types of Interface

**Subroutine**: Separately compiled into object code that can be dynamically linked to the R process. The interface calls the routine, usually via `.Call()`. Rcpp for C++ is the paradigm.
Types of Interface

**Subroutine:** Separately compiled into object code that can be dynamically linked to the R process. The interface calls the routine, usually via `.Call()`. **Rcpp** for C++ is the paradigm.

**Embedded:** Also dynamically linked to R, but with an evaluator for the server language. One subroutine is called from R to initialize, then one or more to send commands/expressions to the evaluator.
Types of Interface

**Subroutine:** Separately compiled into object code that can be dynamically linked to the R process. The interface calls the routine, usually via `.Call()`. **Rcpp** for C++ is the paradigm.

**Embedded:** Also dynamically linked to R, but with an evaluator for the server language. One subroutine is called from R to initialize, then one or more to send commands/expressions to the evaluator.

**Connected:** The server is usually a separate process. R communicates by writing to and reading from connections; the server parses the expressions, evaluates them and writes back the result.
Types of Interface

**Subroutine:** Separately compiled into object code that can be dynamically linked to the R process. The interface calls the routine, usually via `.Call()`. **Rcpp** for C++ is the paradigm.

**Embedded:** Also dynamically linked to R, but with an evaluator for the server language. One subroutine is called from R to initialize, then one or more to send commands/expressions to the evaluator.

**Connected:** The server is usually a separate process. R communicates by writing to and reading from connections; the server parses the expressions, evaluates them and writes back the result.

Embedded interfaces usually win for performance. But connected interfaces are flexible (e.g., JavaScript in a browser; Julia with tasks).
A Unified Approach to Language Interfaces

What we want:

1. R end-user programming should be transparent to the interface for application package users.
A Unified Approach to Language Interfaces

What we want:

1. R *end-user programming* should be transparent to the interface for application package users.

2. For *application package* developers, defining the interface should use language features to automate function and class definition.
A Unified Approach to Language Interfaces

What we want:

1. R end-user programming should be transparent to the interface for application package users.

2. For application package developers, defining the interface should use language features to automate function and class definition.

3. The potential server language software should be unrestricted: any function or method in the server language is a candidate for use, regardless of the class or data type of the arguments or of the value returned.
A Unified Approach to Language Interfaces

What we want:

1. R *end-user programming* should be transparent to the interface for application package users.

2. For *application package* developers, defining the interface should use language features to automate function and class definition.

3. The potential server language software should be unrestricted: *any function or method* in the server language is a candidate for use, regardless of the class or data type of the arguments or of the value returned.

4. Similarly on the R side, one can communicate to or return from the server an object belonging to *any class in R*. 
The Design

One package (XR) has the language-independent classes and functions. Other packages (XRJulia, XRPython, etc.) specialize that.

1. **End-user programming**: Proxy objects, functions and classes are used like “pure” R versions.
The Design

One package (XR) has the language-independent classes and functions. Other packages (XRJulia, XRPython, etc.) specialize that.

1. **End-user programming**: Proxy objects, functions and classes are used like “pure” R versions.

2. **Application package**: Proxy functions and classes are defined directly from server language meta-data.
The Design

One package (XR) has the language-independent classes and functions. Other packages (XRJulia, XRPython, etc.) specialize that.

1. **End-user programming**: Proxy objects, functions and classes are used like “pure” R versions.

2. **Application package**: Proxy functions and classes are defined directly from server language meta-data.

3. **Any server function**: if the value is not one of a few simple types, the interface returns a proxy object to R, automatically embedded in a proxy class if defined.
The Design

One package (XR) has the language-independent classes and functions. Other packages (XRJulia, XRPython, etc.) specialize that.

1. *End-user programming*: Proxy objects, functions and classes are used like “pure” R versions.

2. *Application package*: Proxy functions and classes are defined directly from server language meta-data.

3. *Any server function*: if the value is not one of a few simple types, the interface returns a proxy object to R, automatically embedded in a proxy class if defined.

4. *Any R class*: Server languages use a convention for representing any R object in a language-independent form.