Abstract

We describe a package for both R & S which allows expressions in that language to directly invoke Perl sub-routines and object methods directly. This provides a simple way of making network capabilities available to S, as well as fast regular expression matching, etc. The interface between R/S and Perl is achieved by running an embedded Perl interpreter within the R/S session. This gives us direct access to any existing Perl sub-routines and modules and avoids the use of expensive invocations of Perl as a sub-process that can only communicate with R/S via strings. The interface (will) allows for R and Perl computations to be mixed.

This is intended to provide information for the user who is interested in not only knowing about the available functions in the R/S-Perl interface, but also how they might be used most effectively and what underlies them. A collection of annotated examples will provide a quicker way to get started.

Perl is a wonderful complementary tool to use with R and S. There are certain tasks that are perfectly suited to the Perl language and others that one naturally considers expressing in the S-language because of its. Certain data structures (e.g. trees) are more naturally expressed and manipulated in a language that supports references such as Perl than in S. But S offers many advantages over Perl, but in different aspects. We want the best of both to be available to us as users of either. Additionally, the sets of modules or packages that each provides are quite different due to the different user groups. As statisticians analyze and interact with more computer and network data than previously, having tools such as DNS, FTP, HTTP, SNMP, Dates/Calendars, encryption, etc. integrated with the statistical analysis environment/language makes the overall task easier.

Having a Perl interpreter that is persistent within the S session means that one need not execute Perl scripts in sub-processes by invoking an external Perl session. This persistence gives us a slight increase in efficiency as we avoid spawning new processes. More importantly, however, it means that we can use Perl in much the same way we use R/S interactively. We can store intermediate computations and then decide what to do next, availing of these previously computed values. Also, we do not have to communicate in terms of strings. The commands we invoke need not be passed to Perl in strings or files and the results are objects, not textual representations of objects. This gives us much more freedom to express ourselves and simplify many constructs.

An immediate consequence of having Perl embedded within S/Splus is that we have access to network connections. We never added sockets to S4. With Perl, we have a rich set of communication facilities: sockets, HTTP, FTP, etc. This and the Java and CORBA connections allow us to understand how we can interact to greater effect with other applications and subject domains.

As with all general interfaces (such as *.C()*, *.Call()*, *.Fortran()*, *.Java()*), not only do we get easy access to any of the existing code in that language, but also to any that is created in the future. The cost of experimenting with a new package is greatly reduced. One does not need to have a collection of wrappers or interfaces that communicate with those facilities. Instead, they can be dynamically discovered and called directly. Best of all, a generic interface like this allows people to do things that we might never have thought of. Have fun and let us know of how it is useful and how it is not!

1 Overview

The package allows one to create one or more embedded and persistent (through the session) Perl interpreters. It is very similar in spirit to the Java package for R and S. It provides access to the Perl interpreter from the S-language in the following manners.

Expressions One can evaluate Perl expressions or scripts by passing them as strings.

Files One can (in the near future) direct the Perl intepreter to run a Perl script contained in a file.

Get One can retrieve top-level variables directly.

Sub-Routines In the same way that the *.C()* and *.Fortran()* functions can invoke routines in these two languages, one can invoke Perl sub-routines by specifying the name of the routine and the arguments to that routine in the form of R/S values.

Methods One can invoke methods of Perl objects in the same way that Perl sub-routines are invoked by specifying an additional Perl object whose method is to be invoked.

One can even have multiple Perl interpreters at any time and send different commands to each. These are not necessarily executing in different threads and hence concurrently. It just allows the programmer to manage name-spaces and computations in a different way that can sometimes be more convenient.

Perl is a curious language. It relies heavily on automatic coercion based on the way a variable is used. For example, we might have an array

```
[]
   @a = (1,2,3);
```

We can treat this same object as a scalar by referring to the variable a as `$a` rather than as an array `@a`. Due to the nature the R/S-Perl interface, this contextual information is not available. When we ask for the value of a variable, we cannot use the Perl syntax to specify whether a scalar, array or hash is intended. As a result, we specify the intended type as arguments to some of the R/S functions that define the R/S-Perl interface.

2 Inter-System Object Conversion

The critical aspect of any interface between two systems such as R/S and Perl is how objects in one system are transferred to the other. The need to communicate R objects as arguments to function calls in Perl and to return the results of these calls from Perl to R is obvious.

The mapping of primitive objects between the two systems (R/S and Perl) is reasonably trivial and obvious. In other words, when we pass a string from R to Perl in subroutine call, it is converted to a Perl scalar of internal type PV, and users know this as a Perl string.

integer	IV
character	PV
numeric	NV
logical	IV (will be TRUE and FALSE SVs)

It is natural to map S language vectors to Perl arrays. So a numeric vector containing three values such as

```
[]
  c(1, 2, 3)
```

is converted from R to Perl by creating a Perl array and converting each element of the R object according to table 2 above.

Converting Perl arrays to R is slightly more complicated. This is because the elements of Perl arrays need not be of the same type (as far as I know). So we might have a Perl array such as

```
[]
 @a= (1, "abc", 2);
```

The natural representation of this in R/S is a *list* and so we do this. At present, we make no effort to recognize that the elements are of the same type and so can be simplified to one of the primitive vector types (integer, numeric, character or logical).

Named vectors in R/S are naturally represented as hash table or associative arrays in Perl. For example,

```
[]
  x <- c(a=1, b=2, c=3)
```

can be easily understood in Perl as equivalent to

```
[]
  %x = (’a’, 1, ’b’, 2, ’c’, 3)
```

The conversions of these types are not done by creating equivalent Perl or R expressions as strings and evaluating them in the system to which the object is being converted. This would be very problematic to manage name conflicts, and just to generate the expressions. Instead, these primitive types are converted directly in C code that provides the glue between the two systems, defining the interface.

2.1 Non-Primitive Object Conversion

As with the Java and CORBA packages, we endeavor to leave non-primitive (i.e. class objects) in the language in which they are defined. We transfer a reference in Perl to R by storing it in an internal (i.e. managed in C and not in the Perl namespace) Perl hash table and returning an R/S object that contains sufficient information to resolve that object. In other words, we create an object that contains the key used to store the object in the Perl hash table, an identifier for the table. To allow R and S users to more readily understand manage the object, we also include the Perl class or type of the referenced object when known and also the process identifier (the R process id) in which the object was created and makes sense.

One can operate on these Perl reference objects from within R by invoking methods on them, passing them as arguments to other Perl methods and sub-routines. As they are passed across the R/S-Perl interface, the R/S reference objects are resolved into the Perl values from which these references were generated.

3 Evaluation Perl Expressions

With those preliminaries out of the way, we are ready to actual use the Perl interpreter.

The most obvious and, at times, simplest way to interact with Perl is to have it evaluate a string. We can pass to the Perl interpreter a command presented to it as a legal Perl script in the form of a string. This is much like the way we would call Perl using its command line argument `-e`.

Let’s look at some simple examples. The first expression just prints the number 10.

```
[]
invisible(.PerlExpr(’printf "10\\n";’))
```

The next example prints the string `Hello to R user` *(your login)* `from within Perl`, substituting the actual value for your login. It does this by retrieving it from the environment variables accessible in Perl via the ENV associative array.

```
[]
invisible(.PerlExpr(’printf "Hello to R user ($ENV{\’USER\’}) from within Perl\\n";’))
```

Note how we had to escape the quotes ‘’ surrounding `USER`. To me this is ugly, complicated and very error-prone. It is just one argument against communicating via strings. We’ll see more.

Now, we can also perform assignments in these scripts. Here we create an array containing three strings. a is now a regular Perl variable and persists after this expression is evaluate. Therefore, it is available to use in other scripts/commands. We can also retrieve the values of Perl variables directly from andinto R, as we see in the next line – PerlGet(”a”, array=T). This should return a list of length 3 whose elements are the individual strings in a. We will discuss *.PerlGet()* below.

```
[]

.PerlExpr(’@a = ("a","b","cde");’)
.PerlGet("a", isArray = T)
```

Recall that the interpreter is persistent. This means that any assignments one makes within this string will overwrite existing variables with the same name and will be available to us in future calls to the interpreter. For instance, in the code above, a variable a that existed before we executed the second expression will be discarded and all future references to a will see the array of length three.

This is both good and bad. A problem arises if you assign a value to a Perl variable named foo and then call a function that I have written that also assigns to foo. When my function returns, your value of foo is lost. At best, you will get an error (unlikely). At worst, you will get the wrong answer and not know! This is another argument against using strings to communicate between systems: name space conflicts. We will see that a more structure way to manage the computations is to call methods and sub-routines directly and manage the name space of variables (i.e. assignments) in R rather than Perl.

We should also mention that the scripts can contain more than one expression. We can pass any legal Perl script to the interpreter. This can define its own sub-routines, classes, etc. and load packages, and so on.

In this example, we define an R function that allows one to perform regular expression replacements in a string using the Perl =˜ operator. (Of course, R and S have their own built-in versions of these. This is presented here only for illustration.)

We create the Perl expression in R by pasting together the different strings. We want to end up with something like `$tmp = \textit{your string} ; $tmp =~ s/pattern/replacement/g; $tmp;` So there are three perl expressions within the single script.

```
[]
perlSub <-
function(string, pat, with) {
  tmp <- paste("s/",pat,"/",with,"/g",sep="")
  tmp <- paste("$tmp =~",tmp,";")
  tmp <- paste(paste("$tmp =\"", string, "\";",sep=""), tmp, "$tmp;")

 .PerlExpr(tmp)
}
```

To avoid conflicts with other functions (Perl or R) using $tmp as a variable, we can define a sub-routine and call it.

```
[]
perlSub <-
function(string, pat, with) {
 tmp <- "sub RSsubstitute { my ($tmp, $pat, $with) = @_; $tmp =~ s/$pat/$with/g; return $tmp;}"

 tmp1 <- paste("RSsubstitute(\"",string,"\",\"", pat,"\",\"", with,"\");", sep="")
 tmp <- paste(tmp, tmp1, sep="\n")

 .PerlExpr(tmp)
}
```

In this example, we are defining the `RSsubstitute()` each time the function is called. We can avoid this using the R/S function *.PerlExists()*. This allow us to determine if a Perl variable (of a particular type) is defined.

```
[]
```

```
perlSub <-
function(string, pat, with) {
 if(.PerlExists("RSsubstitute", "CV")) {
   tmp <- "sub RSsubstitute { my ($tmp, $pat, $with) = @_; $tmp =~ s/$pat/$with/g; return $tmp;}"
 } else
   tmp <- ""

 tmp1 <- paste("RSsubstitute(\"",string,"\",\"", pat,"\",\"", with,"\");", sep="")
 tmp <- paste(tmp, tmp1, sep="\n")

 .PerlExpr(tmp)
}
```

This is only important if the fixed part of the script becomes large.

The the issue of escaping quotes and the complexity of creating the Perl script string in the examples above illustrated the potential difficulties of creating the string in R to pass to Perl. When we have to programmatically generate any such string, we will probably choose to use text connections in S4/Splus5/Splus6. In R, one would naturally use *paste()*. Since one should try to write code that works in both systems so as to make the code more useful, one would probably choose *paste()* until "*we*" implement connections in R. However, *paste()* can be cumbersome in some situations. Handling strings is not S's forte. Actually, it is Perl's. So we have a bootstrapping issue.

There are two possible solutions or approaches to this issue of creating Perl scripts. One is to generate the script by writing pieces of it to a file rather than accumulating these in a string. This is practical but also complicated and inflexible. The alternative is to not create the script at all, but to call the relevant Perl functions directly from within R. Rather than represent the R values which are to be arguments to the Perl routine as strings, we can call that routine directly and allow the R/S-Perl interface to translate the R values directly in memory. This preservers the semantics and is simpler for the user. Numeric values are translated to numeric values rather than integers and objects remain as objects.

We will quickly look at how we use the first approach and tell Perl to invoke a script.

4 Evaluating the Contents of a Perl File: A Perl script

Suppose we already have self-contained Perl script residing in a file. Then we can execute that script by passing the file name to the R/S function *.PerlFile()*. This parses and evaluates that script. Any assignments, definitions, etc. processed during the evaluation of this script are available to us after that script has completed.

If you take a look at the file `tests/method.pl` that is provided within the R/S package distribution, you will see that it defines a Perl class named Mine and then uses it. It creates an instance of this class and calls two of its methods wich print to Perl's standard output (by default the same as R). We can execute this script as follows.

```
[]
  .PerlFile(system.file("tests/method.pl",pkg="RSPerl")
```

Note that we can access the variable `$a` and the Mine class after the script has terminated.

```
[]
 ref <- .PerlGet("a")
 ref$Display(1)
 ref <- .PerlNew("Mine", 'a','b','c')
 ref$Display(1)
```

Note that one has to be aware of slight differences between using embedded Perl and the regular standalone version. If a script (either in a file or specified as a string) uses other packages.

If this script needs C-level Perl extensions, you should specify the name of a C routine that will initialize these. This can be done automatically. See .PerlInit.

5 Calling Sub-Routines

```
[]

.Perl("Sum", 1,2,3,4)
```

6 Retrieving Perl Objects

.PerlGetArray .PerlGetTable .PerlGet

```
[]
```

7 Creating Perl Arrays and Tables

.PerlNewTable() and *.PerlNewArray()*

```
[]
```

.PerlAssign()

```
[]
```

```
One can apply a filter to the elements being retrieved.  In this
example, we call the Reverse sub-routine defined in
\file{tests/RInit.pl}.
```

```
[]
.PerlExpr('@x = ("abc","def", "ghi");')
.PerlGetArray("x", apply="Reverse")
```

8 Operator Overloading

If one obtains a reference to a Perl array via a call to

```
 .PerlGetArray(name, .convert=F)
```

then one can use the regular R/S syntax to extract elements of the array.

```
[]
.PerlExpr('@a=("a","b","c");')
ref <- .PerlGetArray("a", .convert = F)

ref[1,2]
[[1]]
[1] "a"

[[2]]
[1] "b"
```

9 Static/Class Methods

We use the same *.Perl()* to invoke static methods. However, we specify the name of the class in the call via the *ref* argument. Rather than being a reference to a Perl object, this is given as the name of the Perl class whose static method is to be invoked. As an example, we can call the PrintID in the class Mine. (This is defined in the file `method.pl` in the `examples/` directory of the distribution.) The method expects no arguments and so we specify the name of the method and the name of the class in which the method is defined.

```
[]
 .PerlFile(system.file("examples/method.pl", pkg="Java"))

 .Perl("PrintID", ref="Mine", array=F)
```

10 Creating Objects

Given a class definition with a method new, we can invoke that method as a static class method.

Not quite working. Needs some thinking. There is likely to be a C routine

```
[]
.Perl("new", arg1, arg2,...,ref="className")
```

11 Retrieving Values

One can get the value of a variable in the top-level context or packages using the *.PerlGet*()* functions. As with all Perl operations, one must indicate the type of the variable so that it can be coerced to the appropriate form (array, scalar, hash, etc.) This is done by selecting the appropriate function (rather than an argument to a single function).

```
[]
.PerlGetTable("ENV")
```

```
[]
.PerlExpr("@a=('x','y','z');")
```

```
[1] 3
> .PerlGetArray("a")
[[1]]
[1] "x"

[[2]]
[1] "y"

[[3]]
[1] "z"

>
```

12 Reflectance

One can get a list of the variables within a package using *.PerlStashInfo()*.

```
[]
names(.PerlStashInfo("main"))

.PerlPackage("News::NNTPClient"))
names(.PerlStashInfo("News::NNTPClient"))
```

13 Calling Code Directly, not by Name

The following shows how we can get a handle on a sub-routine or method and call it directly, rather than by name.

```
[]
pj <- .PerlGetCode("Join")
.Perl(pj, "--", "abc","def")
```

The benefit of doing this is again to avoid clutter and conflict in the name-space. One can get create anonymous routines and store references to them. All other code is unaware of their existence, but we can call them in the following manner.

```
[]

r <- .PerlExpr('sub { my ($l) = @_; print "$l\n"; return 1;}')
.Perl(r, "This is a test")
```

This is not an entirely technical and pedantic point. Many of us have experience with name conflicts in S. Programmers sometimes circumvent apparent difficulties with the scoping rules in S and assign variables that are to be shared by different functions in frame 0. This happens frequently in the modelling code. The potential for two developers to use the same variable name may be small. It is non-zero however and the improbability of the event occurring means that we rarely think of it when spending numerous hours debugging the problem.

14 Garbage Collection

Basically, by embedding a persistent Perl interpreter within an R/S session, we have two interactive workspaces in existence at any time. As with R/S by itself, one can assign the results of (intermediate) computations at the top-level so that they can be used later. Unfortunately, these can be large and consume resources, especially memory. Having a Perl interpreter open at the same time means that one can do this in more ways. One can assign values to Perl variables via Perl expressions evaluated via *.PerlExpr()* and *.PerlFile()*, and also implicitly within the internal *PerlReference* objects that are returned (anonymously) by these calls and others.

Just as with R and S, the system cannot discard these objects automatically. Each interpreter cannot determine when you, the user, has finished operating on these objects, so it must keep them for potential future computations. As a result, the garbage collection – the discarding of no longer needed objects – is up to the user. One can undefine Perl variables via the *.PerlUndef()* function. Also, one can discard anonymous *PerlReference* objects via the *.PerlDiscard()* function. In each case, the Perl object is discarded and its resources can be reclaimed.

To determine what Perl variables are defined, one can use the *.PerlObjects()* function. Similarly, to get a list of all the *PerlReference* objects currently in the internal table, one can use the function *.PerlReferenceObjects()*. To simply get the number of entries in this table, one can use the function *.PerlReferenceCount()*. This returns both the current number of entries and also the total number of reference objects that have been created in the session.

15 Arrays and Hash Tables

.PerlNewArray() and *.PerlNewTable()*

The subsetting and element assignment operators

```
[]
 x[1] <- 1
 x[1]

 x["a"] <- 1
 x["a"]
```

.PerlLength()

.PerlClear()

.PerlNames() and *names()*.

16 Style

Should one paste together a Perl command in the form of a string or call the Perl sub-routine or method with arguments that are given by R objects? My feeling is that the latter is greatly preferrable. There are several reasons. One is that it insulates the function call from being Perl-dependent. It is possible for one to replace this code with a call to a CORBA operation or a Java method by changing only the reference to the *.Perl()* function. A second reason relates to the difficulty associated with creating the string to represent the Perl expression. In many cases, one can simple use *paste()* to create the string. However, in other cases, the R values that one wishes to represent in the Perl expression do not have a simple string representation. This is why we want to allow them to be exported as references to R objects whose methods are implemented by R functions. When we use strings, this is hard. When we use real objects, we have the flexibility to allow such foreign/indirect references.

17 Using Extensions That Also Use C Code

When one first attempt to load the NNTPClient (see `http://www.`), an error similar to the following may appear.

```
[]
Can't load module Socket, dynamic loading not available in this perl.
  (You may need to build a new perl executable which either supports
  dynamic loading or has the Socket module statically linked into it.)
 at /usr/lib/perl5/site_perl/5.005/News/NNTPClient.pm line 6
BEGIN failed--compilation aborted at /usr/lib/perl5/site_perl/5.005/News/NNTPClient.pm line 6.
BEGIN failed--compilation aborted at (eval 1) line 1.
```

This indicates that there is a problem loading the compiled C-level code relating to the Socket module which is used by the News::NTTPClient module. To fix this, we must tell the embedded Perl interpreter how to communicate with that code. We do this by supplying a value for the argument *extensions* in the call to *.PerlInit()* (or alternatively in a call to *.PerlFile()*). This argument should be the name of a C-level routine that tells Perl how to find the C libraries of the necessary modules.

Perl provides a tool to generate this code. The following command generates a C file named `xsinit.c` which defines the routine `xs_init()` with the necessary commands to tell Perl how to dynamically load the C code for the IO and Socket modules. Other

```
[]
perl -MExtUtils::Embed -e xsinit -- -o xsinit.c -std IO Socket
```

One can add modules to this when compiling the Perl package. This can be done by adding to elements to the PERL_MODULES variable in `GNUmakefile`. Alternatively, one can specify these on the command line and regenerate `xsinit.c`.

Note, that the shared library associated with that module must be available to the Perl engine at run time so that symbols can be resolved. This is a little trickier than when running Perl as a stand-alone executable. Accordingly, for the present, we just add these libraries to the link command in the Makefile. For example, we find where `Socket.so` is located on the system and include that in the list of objects identified by PERL_MODULE_SOS.

In the future, we will find a way to make this more dynamic and to be able to dynamically load these modules from R itself and make them available to Perl from there. This involves playing with different flags.

References

[1] The Perl Module List. URL, http://www.cpan.org/modules/00modlist.long.htmlPart2-ThePerl5M, October 2000.

[2] Sriram Srinvasan. *Advanced Perl Programming.* The Perl Series. O'Reilly & Associates, Inc., 1997.

[2] [1]
[?]