A Future for R: Common Issues with Solutions

In the ideal case, all it takes to start using futures in R is to replace select standard assignments (`<-`) in your R code with future assignments (`%<-%`) and make sure the right-hand side (RHS) expressions are within curly brackets (`{ ... }`). Also, if you assign these to lists (e.g. in a for loop), you need to use a list environment (`listenv`) instead of a plain list.

However, as show below, there are few cases where you might run into some hurdles, but, as also shown, they are often easy to overcome. These are often related to global variables.

If you identify other cases, please consider reporting them so they can be documented here and possibly even be fixed.

Issues with globals and packages

Missing globals (false negatives)

If a global variable is used in a future expression that *conditionally* overrides this global variable with a local one, the future framework fails to identify the global variable and therefore fails to export it, resulting in a run-time error. For example, although this works:

```
library(multisession)

reset <- TRUE
x <- 1
y %<-% { if (reset) x <- 0; x + 1 }
y
## [1] 1
```

the following does *not* work:

```
reset <- FALSE
x <- 1
y %<-% { if (reset) x <- 0; x + 1 }
y
## Error: object 'x' not found
```

Comment: The goal is to in a future version of the package detect globals also in expression where the local-global state of a variable is only known at run time.

do.call() - function not found

When calling a function using `do.call()` make sure to specify the function as the object itself and not by name. This will help identify the function as a global object in the future expression. For instance, use

```
do.call(file_ext, list("foo.txt"))
```

instead of

```
do.call("file_ext", list("foo.txt"))
```

so that `file_ext()` is properly located and exported. Although you may not notice a difference when evaluating futures in the same R session, it may become a problem if you use a character string instead of a function object when futures are evaluated in external R sessions, such as on a cluster. It may also become a problem with futures evaluated with lazy evaluation if the intended function is redefined after the future is resolved. For example,

```
> library("future")
> library("listenv")
> library("tools")
> plan(sequential)
> pathnames <- c("foo.txt", "bar.png", "yoo.md")
> res <- listenv()
> for (ii in seq_along(pathnames)) {
+   res[[ii]] %<-% do.call("file_ext", list(pathnames[ii])) %lazy% TR
+ }
> file_ext <- function(...) "haha!"
> unlist(res)
[1] "haha!" "haha!" "haha!"
```

Missing packages (false negatives)

Occassionally, the static-code inspection of the future expression fails to identify packages needed to evaluated the expression. This may occur when an expression uses S3 generic functions part of one package whereas the required S3 method is in another package. For example, in the below future generic function `[` is used on data.table object `DT`, which requires S3 method `[.data.table` from the data.table package. However, the future and global packages fail to identify data.table as a required package, which results in an evaluation error:

```
> library("future")
> plan(multisession)

> library("data.table")
> DT <- data.table(a = LETTERS[1:3], b = 1:3)
> y %<-% DT[, sum(b)]
> y
Error: object 'b' not found
```

The above error occurs because, contrary to the master R process, the R worker that evaluated the future expression does not have data.table loaded. Instead the evaluation falls back to the `[.data.frame` method, which is not what we want.

Until the future framework manages to identify data.table as a required package (which is the goal), we can guide future by specifying additional packages needed:

```
> y %<-% DT[, sum(b)] %packages% "data.table"
> y
[1] 6
```

or equivalently

```
> f <- future(DT[, sum(b)], packages = "data.table")
> value(f)
[1] 6
```

Non-exportable objects

Certain types of objects are tied to a given R session and cannot be passed along to another R process (a “worker”). For instance, if you create a file connection,

```
> con <- file("output.log", open = "wb")
> cat("hello ", file = con)
> flush(con)
> readLines("output.log", warn = FALSE)
[1] "hello "
```

it will not work when used in another R process(*). If we try, the result is “unknown”, e.g.

```
> library("future")
> plan(multisession)
> f <- future({ cat("world!", file = con); flush(con) })
> value(f)
NULL
> close(con)
> readLines("output.log", warn = FALSE)
[1] "hello "
```

In other words, the output `"world!"` written by the R worker is completely lost.

The culprit here is that the connection uses a so called *external pointer*:

```
> str(con)
Classes 'file', 'connection'  atomic [1:1] 3
  ..- attr(*, "conn_id")=<externalptr>
```

which is bound to the main R process and makes no sense to the worker. Ideally, the R process of the worker would detect this and produce an informative error message, but as seen here, that does not always occur.

To help avoiding these mistakes, the future framework provides a mechanism (*in beta*) can help detect non-exportable objects by setting:

```
> options(future.globals.onReference = "warning")
```

which will result in a warning in the above example:

```
> f <- future({ cat("world!", file = con); flush(con) })
Warning in FALSE :
  Detected a non-exportable reference ('externalptr') in one of the g
('con' of class 'file') used in the future expression
```

To be more conservative, we can also tell it to produce an error:

```
> options(future.globals.onReference = "error")
> f <- future({ cat("world!", file = con); flush(con) })
Error in FALSE :
  Detected a non-exportable reference ('externalptr') in one of the g
('con' of class 'file') used in the future expression
```

Below are few examples of commonly used packages that produce non-exportable objects. If you find other examples, please report back.

(*) The exception *may* be if you use *forked* R processes, e.g. `plan(multicore)`.

However, such attempts to work around the underlying problem, which is non-exportable objects, should be avoided and considered non-stable. Moreover, such code will certainly fail to parallelize on other future backends.

Example: Non-exportable objects by the xml2 package

Another example is XML objects of the xml2 package, which may produce evaluation errors (or just invalid results depending on how they are used), e.g.

```
> library("future")
> plan(multisession)
> library("xml2")
> xml <- read_xml("<body></body>")
> f <- future(xml_children(xml))
> value(f)
Error: external pointer is not valid
```

The future framework can help detect this *before* sending off the future to the worker;

```
> options(future.globals.onReference = "error")
> f <- future(xml_children(xml))
Error in FALSE :
  Detected a non-exportable reference ('externalptr') in one of the g
('xml' of class 'xml_document') used in the future expression
```

Example: Non-exportable objects by the Rcpp package

A third example is Rcpp which allow us to easily create R functions that are implemented in C++, e.g.

```
Rcpp::sourceCpp(code = "
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int my_length(NumericVector x) {
    return x.size();
}
")
```

so that:

```
> x <- 1:10
> my_length(x)
[1] 10
```

However, since this function uses an external pointer internally, we cannot pass it to another R process:

```
> library("future")
> plan(multisession)
> n %<-% my_length(x)
> n
Error: NULL value passed as symbol address
```

We can detect / protect against this using:

```
> options(future.globals.onReference = "error")
> n %<-% my_length(x)
Error in FALSE :
  Detected a non-exportable reference ('externalptr' of class 'Native
in one of the globals ('my_length' of class 'function') used in the f
expression
```

Example: Non-exportable objects by the rJava package

Here is an example that show how rJava objects cannot be used when exported to external R processes.

```
> library("future")
> plan(multisession)

> library("rJava")
> .jinit() Initialize Java VM on master

> Double <- J("java.lang.Double")
> d0 <- new(Double, "3.14")
> print(d0)
[1] "Java-Object{3.14}"

> f <- future({
    .jinit() ## Initialize Java VM on worker
    new(Double, "3.14")
  })
> d1 <- value(f)
> print(d1)
## [1] "Java-Object<null>"
```

Although no error is produced, we see that the value `d1` is a Java NULL Object. As before, we can catch this by using:

```
> options(future.globals.onReference = "error")
> f <- future({
    .jinit() ## Initialize Java VM on worker
    new(Double, "3.14")
> })
Error in FALSE :
  Detected a non-exportable reference ('externalptr') in one of the g
  ('Double' of class 'jclassName') used in the future expression
```

Example: Non-exportable objects by the rlang package

Yet another example comes from the rlang package. This example is somewhat twisted - it uses `value(future(x))` which is quite meaningless by itself- but it is a minimal example of showing what can happen:

```
> library("future")
> plan(multisession)

> library("dplyr")  ## relies on 'rlang'
> iris <- as_tibble(iris)
> res <- mutate(iris, mod = value(future({
    list(lm(Species ~ Sepal.Length, data = .))
  })))
Error in mutate_impl(.data, dots) :
  Evaluation error: NULL value passed as symbol address.
```

In this case the mistake is caught by dplyr's `mutate_impl()`, but to be on the safe side, we could have used:

```
> options(future.globals.onReference = "error")
> res <- mutate(iris, mod = value(future({
    list(lm(Species ~ Sepal.Length, data = .))
  })))
Timing stopped at: 0.004 0 0.001
Error in mutate_impl(.data, dots) :
  Evaluation error: Detected a non-exportable reference ('externalptr
  of class 'RegisteredNativeSymbol') in one of the globals ('~' of
  class 'function') used in the future expression.
```

Trying to pass an unresolved future to another future

It is not possible for a future to resolve another one unless it was created by the future trying to resolve it. For instance, the following gives an error:

```
> library("future")
> plan(multiprocess)
> f1 <- future({ Sys.getpid() })
> f2 <- future({ value(f1) })
> v1 <- value(f1)
[1] 7464
> v2 <- value(f2)
Error: Invalid usage of futures: A future whose value has not yet bee
 can only be queried by the R process (cdd013cb-e045-f4a5-3977-9f064c
 1276 on MyMachine) that created it, not by any other R processes (55
 -bace-c50d-6c7a23ddb5a3; pid 2352 on MyMachine): {; Sys.getpid(); }
```

This is because the main R process creates two futures, but then the second future tries to retrieve the value of the first one. This is an invalid request because the second future has no channel to communicate with the first future; it is only the process that

created a future who can communicate with it(*).

Note that it is only *unresolved* futures that cannot be queried this way. Thus, the solution to the above problem is to make sure all futures are resolved before they are passed to other futures, e.g.

```
> f1 <- future({ Sys.getpid() })
> v1 <- value(f1)
> v1
[1] 7464
> f2 <- future({ value(f1) })
> v2 <- value(f2)
> v2
[1] 7464
```

This works because the value has already been collected and stored inside future `f1` before future `f2` is created. Since the value is already stored internally, `value(f1)` is readily available everywhere. Of course, instead of using `value(f1)` for the second future, it would be more readable and cleaner to simply use `v1`.

The above is typically not a problem when future assignments are used. For example:

```
> v1 %<-% { Sys.getpid() })
> v2 %<-% { v1 }
> v1
[1] 2352
> v2
[1] 2352
```

The reason that this approach works out of the box is because in the second future assignment `v1` is identified as a global variable, which is retrieved. Up to this point, `v1` is a promise ("delayed assignment" in R), but when it is retrieved as a global variable its value is resolved and `v1` becomes a regular variable.

However, there are cases where future assignments can be passed via global variables without being resolved. This can happen if the future assignment is done to an element of an environment (including list environments). For instance,

```
> library("listenv")
> x <- listenv()
> x$a %<-% { Sys.getpid() }
> x$b %<-% { x$a }
> x$a
[1] 2352
> x$b
Error: Invalid usage of futures: A future whose value has not yet bee
 can only be queried by the R process (cdd013cb-e045-f4a5-3977-9f064c
 1276 on localhost) that created it, not by any other R processes (2c
 -7a05-1373-e1b20022e4d8; pid 7464 on localhost): {; Sys.getpid(); }
```

As previously, this can be avoided by making sure `x$a` is resolved first, which can be one in various ways, e.g. `dummy <- x$a`, `resolve(x$a)` and `force(x$a)`.

Footnote: (*) Although sequential futures could be passed on to other futures part of the same R process and be resolved there because they share the same evaluation process, by definition of the Future API it is invalid to do so regardless of future type. This conservative approach is taken in order to make future expressions behave consistently regardless of the type of future used.

Miscellaneous

Clashes with other packages

Sometimes other packages have functions or operators with the same name as the future package, and if those packages are attached *after* the future package, their objects will mask the ones of the future package. For instance, the igraph package also defines a `%<-%` operator which clashes with the one in future *if used at the prompt or in a script* (it is not a problem inside package because there we explicitly import objects in a known order). Here is what we might get:

```
> library("future")
> library("igraph")

Attaching package: 'igraph'

The following objects are masked from 'package:future':

    %<-%, %->%

The following objects are masked from 'package:stats':

    decompose, spectrum

The following object is masked from 'package:base':

    union

> y %<-% { 42 }
Error in get(".igraph.from", parent.frame()) :
  object '.igraph.from' not found
```

Here we get an error because `%<-%` is from igraph and not the future assignment operator as we wanted. This can be confirmed as:

```
> environment(`%<-%`)
<environment: namespace:igraph>
```

To avoid this problem, attach the two packages in opposite order such that future comes last and thereby overrides igraph, i.e.

```
> library("igraph")
> library("future")

Attaching package: 'future'

The following objects are masked from 'package:igraph':

%<-%, %->%

> y %<-% { 42 }
> y
[1] 42
```

An alternative is to detach the future package and re-attach it, which will achieve the same thing:

```
> detach("package:future")
> library("future")
```

Yet another alternative is to explicitly override the object by importing it to the global environment, e.g.

```
> `%<-%` <- future::`%<-%`
> y %<-% { 42 }
> y
[1] 42
```

In this case, it does not matter in what order the packages are attached because we will always use the copy of `future::`%<-%``.

Syntax error: "non-numeric argument to binary operator"

The future assignment operator `%<-%` is a *binary infix operator*, which means it has higher precedence than most other binary operators but also higher than some of the unary operators in R. For instance, this explains why we get the following error:

```
> x %<-% 2 * runif(1)
Error in x %<-% 2 * runif(1) : non-numeric argument to binary operato
```

What effectively is happening here is that because of the higher priority of `%<-%`, we first create a future `x %<-% 2` and then we try to multiply the future (not its value) with the value of `runif(1)` - which makes no sense. In order to properly assign the future variable, we need to put the future expression within curly brackets;

```
> x %<-% { 2 * runif(1) }
> x
[1] 1.030209
```

Parentheses will also do. For details on precedence on operators in R, see Section 'Infix and prefix operators' in the 'R Language Definition' document.

R CMD check NOTEs

The code inspection run by `R CMD check` will not recognize the future assignment operator `%<-%` as an assignment operator, which is not surprising because `%<-%` is technically an infix operator. This means that if you for instance use the following code in your package:

```
foo <- function() {
  b <- 3.14
  a %<-% { b + 1 }
  a
}
```

then `R CMD check` will produce a NOTE saying:

```
* checking R code for possible problems ... NOTE
foo: no visible binding for global variable 'a'
Undefined global functions or variables:
  a
```

In order to avoid this, we can add a dummy assignment of the missing global at the top of the function, i.e.

```
foo <- function() {
  a <- NULL ## To please R CMD check
  b <- 3.14
  a %<-% { b + 1 }
  a
}
```