The Problem

I wish to write a function that can mutate a column of my data using a function which is passed as an input. For example, I want to write a function which can modify an existing `tbl_spark` using `dplyr::mutate()` by adding a column that is the `mean()` of a column. To do this, we will see that we require a different solution depending on whether we are working with a `data.frame` or a `tbl_spark`. For this reason, I am going to define an S3 generic, `mutate_with()`, for which I am going to define two methods, one for `data.frame`s and one for `tbl_spark`s.

Solution For `data.frames`

First of all, let us define our generic. If you are not familiar with S3 generics and how they work, I would recommend checking out the S3 section in the freely available Advanced R book.

```
mutate_with <- function(data, fn, col_in, col_out = "output") {
  UseMethod(generic = "mutate_with", object = data)
}
```

Now that we have our generic in place, we can add our first method for `data.frame`s. This is actually relatively simple as seen below.

```
mutate_with.data.frame <- function(data, fn, col_in, col_out =
"output") {
  dplyr::mutate(.data = data, !!col_out := fn(!!rlang::sym(col_in)))
}
head(
  mutate_with(data = mtcars, fn = mean, col_in = "mpg", col_out =
"mean_mpg")
)
#    mpg cyl disp  hp drat    wt  qsec vs am gear carb mean_mpg
# 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4 20.09062
# 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4 20.09062
# 3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1 20.09062
# 4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1 20.09062
# 5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2 20.09062
# 6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1 20.09062
```

This works because what we are essentially doing is assigning the function `mean()` to the object `fn`.

```
mean
# function (x, ...)
# UseMethod("mean")
#
#
fn <- mean
fn
# function (x, ...)
# UseMethod("mean")
#
```

```
#
```

This means that we can call `fn()` as if we were calling `mean()`.

```
fn(rnorm(100))
# [1] 0.01109211
```

Solution For `tbl_sparks`

Unfortunately the above solution does not work for `tbl_spark`s. To show this, I will first create a Spark connection and upload the `mtcars` data.

```
sc <- sparklyr::spark_connect(master = "local")
mtcars_spark <- dplyr::copy_to(sc, mtcars, "mtcars")
```

Now let's try calling our function with the `mtcars_spark` object.

```
mutate_with.tbl_spark <- mutate_with.data.frame
mutate_with(
  data = mtcars_spark, fn = mean, col_in = "mpg", col_out = "mean_mpg"
)
# Error: org.apache.spark.sql.AnalysisException: Undefined function:
'fn'.
# This function is neither a registered temporary function nor a
permanent
# function registered in the database 'default'.; line 1 pos 85
# ... (additional output has been removed)
```

So we get an error here telling us that we have passed an underfined function, `fn`. This is because in the background, the package {dbplyr} is attempting to translate our code to Spark SQL code to be executed by Spark. To do this, {dbplyr} parses our {dplyr} code, for which there are two parts.

Firstly there is the translation of {dplyr} verbs, e.g. `select()`, `filter()`, `arrange()`, etc. form the basic structure of a SQL query, `SELECT`, `WHERE`, `ORDER BY`, etc.

Secondly there is the translation of the expressions within those verbs. This is the part that is interesting to us for this blog post. First of all we have "known functions" where {dbplyr} takes the functionality that you pass to it and tries to convert it to a SQL equivalent. As a simple example, {dbplyr} will convert the `mean()` function to `AVG`. However {dbplyr} cannot always produce a perfect translation because there are functions which exist in R that do not exist in the varying flavours of SQL, we will refer to these as "unknown functions".

> As a note, there is even functionality in both that cannot be translated, for example the `trim` parameter of the `mean()` function is not available in SQL.

What {dbplyr} does with these unknown functions, therefore, is to leave them "as-is". This means that database functions that are not covered by {dbplyr} can be used directly. For example in Spark we can access and use Hive UDFs.

So what does this all mean? And why does this mean that our function failed? Well it's because when {dbplyr} parses our code, it "sees" the `fn` parameter and it treats it "as-is" since there is no direct conversion to a SQL function defined in {dbplyr}. Equally, there is no function in Spark SQL called `fn()` which is why, when our converted code is sent to Spark, we see the failure.

We can see this in action by rendering the translated output that {dbplyr} gives us.

```
out <- mutate_with(
  data = mtcars_spark, fn = mean, col_in = "mpg", col_out = "mean_mpg"
)
dbplyr::sql_render(out)
 SELECT `mpg`, `cyl`, `disp`, `hp`, `drat`, `wt`, `qsec`, `vs`, `am`,
`gear`, `carb`, fn(`mpg`) AS `mean_mpg`
FROM `mtcars`
```

This means that we must pass the actual function *name*, i.e. `mean`, in place of `fn` at the time that {dbplyr} "sees" our code and attempts to parse and translate it into SQL. For this, we need to use non-standard evaluation semantics.

```
mutate_with.tbl_spark <- function(data, fn, col_in, col_out = "output")
{
  fn <- rlang::enquo(fn)
  dplyr::mutate(
    .data = data, !!col_out := rlang::call2(.fn = !!fn,
rlang::sym(col_in))
  )
}
mutate_with(
  data = mtcars_spark, fn = mean, col_in = "mpg", col_out = "mean_mpg"
)
# # Source: spark [?? x 12]
#      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
mean_mpg
#
#  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
20.1
#  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
20.1
#  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
20.1
#  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
20.1
#  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
20.1
#  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
20.1
#  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
20.1
#  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
20.1
#  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
20.1
# 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
20.1
# # … with more rows
```

So why does this work? Well what we are doing here is "quoting" the parameter `fn` which is the same as saying preventing the evaluation of `fn`. `rlang::enquo()` is used to quote function

arguments such as `fn` and returns what is known as a "quosure". A "quosure" is an object containing both the expression and the environment which that expression comes from. We can "unquote" the "quosure", `fn`, with the `!!` or "bang-bang" operator.

In English this means that we can evaluate and replace inline the captured expression. So what `!!fn` actually reads as is `mean`. The `!!fn` works because it is evaluated *before* any R code is evaluated. In other words, before the code is parsed by {dbplyr}, {rlang} is able to evaluate `!!fn`, replacing it with `mean`. You will sometimes see this being referred to as partial evaluation.

That's a lot of information in two short paragraphs and I'd highly recommend reading the help file (`?rlang::quosure`) if you'd like to know more. Another great resource for learning more about {rlang} is Brodie Gaslam's On Quosures.

We use `rlang::call2()` to build a quoted function call which can be interpreted by {dbplyr}. What this all boils down to is that `rlang::call2(.fn = !!fn, rlang::sym(col_in))` is used to create the following, unevaluated, expression.

```
mean(mpg)
```

So the execution order is as follows:

1. `!!fn` is translated to `mean`, inline.
2. `rlang::call2()` is evaluated, giving `mean(mpg)` (note that we convert our string input to a symbol using `rlang::sym(col_in)`).
3. {dbplyr} translates the code to SQL code and sends it to Spark to be executed.

Note that we can achieve a similar result using `base::bquote()` and `base::eval()`. `bquote()` is base R's equivalent of partial evaluation which creates a quoted expression that we can then evaulate with `eval()`. The difference in syntax is that the parts to be partially evaluated are surrounded by `.()` instead of `!!` as in {rlang}.

```
mutate_with_bquote <- function(data, fn, col_in, col_out = "output") {
  eval(bquote(
    dplyr::mutate(.data = data, !!col_out := .(fn)(rlang::sym(col_in)))
  ))
}
mutate_with_bquote(
  data = mtcars_spark, fn = mean, col_in = "mpg", col_out = "mean_mpg"
)
# # Source: spark [?? x 12]
#      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
mean_mpg
#
#  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
20.1
#  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
20.1
#  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
20.1
#  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
20.1
#  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
20.1
```

```
#  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
20.1
#  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
20.1
#  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
20.1
#  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
20.1
# 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
20.1
# # … with more rows
```

Quoted Function Name

The above could have all been made much simpler by passing a quoted function name, however you may well have reasons you don't wish to do this. For completeness, I present an example of how this can be done.

```
mutate_with.tbl_spark <- function(data, fn, col_in, col_out = "output")
{
  condition <- paste0(fn, "(", col_in, ")")
  dplyr::mutate(.data = data, !!col_out :=
!!rlang::parse_expr(condition))
}
mutate_with(
  data = mtcars_spark, fn = "mean", col_in = "mpg", col_out =
"mean_mpg"
)
# # Source: spark [?? x 12]
#      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
mean_mpg
#
#  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
20.1
#  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
20.1
#  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
20.1
#  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
20.1
#  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
20.1
#  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
20.1
#  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
20.1
#  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
20.1
#  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
20.1
# 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
20.1
```

```
# # … with more rows
```

So we build the expression as a string which we then parse ourselves using `rlang::parse_expr()`.

It is also worth noting that this version works with regular `data.frame`s too.

```
mutate_with.data.frame <- function(data, fn, col_in, col_out =
"output") {
  condition <- paste0(fn, "(", col_in, ")")
  dplyr::mutate(.data = data, !!col_out :=
!!rlang::parse_expr(condition))
}
head(mutate_with(
  data = mtcars, fn = "mean", col_in = "mpg", col_out = "mean_mpg"
))
#    mpg cyl disp  hp drat    wt  qsec vs am gear carb mean_mpg
# 1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4 20.09062
# 2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4 20.09062
# 3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1 20.09062
# 4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1 20.09062
# 5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2 20.09062
# 6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1 20.09062
```

Additional Comments

What is nice about the solution presented in this blog post is that we can still pass any user defined functions. Let's say we have a function written in Scala code and exposed to the Spark cluster. We can still call this function. For example below I am using the Hive UDF, `reverse()`, to return a column with my last name spelt backwards.

```
data <- data.frame(first_name = "nathan", last_name = "eastwood")
data <- dplyr::copy_to(sc, data, "data")
mutate_with(data = data, fn = reverse, col_in = "last_name")
# # Source: spark [?? x 3]
#   first_name last_name output
#
# 1 nathan     eastwood  doowtsae
```

In addition it can be a nice way to get around `R CMD check` issues. Let's say we wanted to use the Spark function `sequence()`.

```
add_sequence <- function(data, length_to, col_out = "seq") {
  dplyr::mutate(.data = data, !!col_out := sequence(1, length_to))
}
```

If we have the above function defined in an R package, we may see the following `R CMD check NOTE`.

```
checking R code for possible problems ... NOTE
add_sequence: possible error in sequence(1,
  length_to): unused argument (length_to)
```

Whereas using the trick shown in this blog, we no longer see this issue.

```
add_sequence <- function(data, fn = sequence, from, to, col_out =
"seq") {
  fun <- rlang::enquo(fn)
  dplyr::mutate(.data = data, !!col_out := rlang::call2(.fn = !!fun,
from, to))
}
```

Of course, I wouldn't recommend defining such a function this way.

Conclusion

To keep things short and sweet, if you wish to pass a function as-is to another function which you then wish to have parsed by dbplyr into SQL code, you must first quote the function with `rlang::enquo()` and then unquote it with `!!`. You can then build your expression with `rlang::call2()` and have this parsed by {dbplyr}.