The rquery R package has several places where the user can ask for what they have typed in to be substituted for a name or value stored in a variable.

This becomes important as many of the rquery commands capture column names from un-executed code. So knowing if something is treated as a symbol/name (which will be translated to a data.frame column name or a database column name) or a character/string (which will be translated to a constant) is important.

**strings/character versus names/symbols**

Let’s take a look at this through small examples. First let’s take a look at the difference between strings and symbols in R.

col\_string <- "x"

col\_name <- as.name(col\_string)

str(col\_string)

## chr "x"

str(col\_name)

## symbol x

Notice, in R a string is different than a symbol.

We can see this difference in rquery where an un-quoted x is treated as a symbol (and therefore is translated to a database column) and a quoted entity is treated as a string (and therefore is translated to a literal or constant, not to a column).

library("rquery")

d <- data.frame(x = c('a', 'b'),

stringsAsFactors = FALSE)

d\_rep <- local\_td(d)

db\_info <- rquery\_db\_info(identifier\_quote\_char = "\_\_IDENTIFIER\_\_",

string\_quote\_char = "\_\_STRING\_CONSTANT\_\_")

# direct use, comparing to a string constant

# probaly not the query we intend as the

# result is going to be empty independent

# of the data.

cat(to\_sql(

d\_rep %.>% select\_rows(., is.na('x')),

db\_info))

## Warning in warn\_about\_filter\_conditions(parsed): rquery::select\_rows:

## expression is.na("x") refers to no columns (so is a constant)

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_22528812269810523566\_0000000000

## WHERE ( ( \_\_STRING\_CONSTANT\_\_x\_\_STRING\_CONSTANT\_\_ ) IS NULL )

(The warning is new to rquery version 1.3.2.)

We take careful note what is marked as "\_\_IDENTIFIER\_\_", versus what is marked as "\_\_STRING\_CONSTANT\_\_". Notice "\_\_IDENTIFIER\_\_" is used in the SQL for table names and column name, and "\_\_STRING\_CONSTANT\_\_" is used for string constants. The above query is probably not what a user intended as we are checking if a user supplied string constant is NA, which is not interesting.

Likely the correct query omits the quote marks from the x.

# direct use, comparing to a column

cat(to\_sql(

d\_rep %.>% select\_rows(., is.na(x)),

db\_info))

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_46294476393859703536\_0000000000

## WHERE ( ( \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_ ) IS NULL )

In the above query we are now comparing an identifier to NULL, which is how SQL expresses comparing the contents of the column named to NULL in a row by row fashion (a useful query).

Or combing the two ideas. We check which rows of the column x have the value "a" as follows.

cat(to\_sql(

d\_rep %.>% select\_rows(., x == 'a'),

db\_info))

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_37960424657179884082\_0000000000

## WHERE \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_ = \_\_STRING\_CONSTANT\_\_a\_\_STRING\_CONSTANT\_\_

**wrapr::let() substitution**

wrapr::let() substitution is designed only to substitute in names as if the user had typed them. It is deliberately not designed to deal with other value substitutions (such as strings, integers, or floating point values). This is intentional and to keep wrapr::let() to one job: adapting NSE (Non-standard interfaces) to accept names as values.

wrapr::let()‘s principle is that there is no reason for wrapr::let() to ever substitute in a value (such as a string or an integer) as normal evaluation of variable names in environments already supplies a better way to do that. The only thing that is hard to substitute in are new symbols, so wrapr::let() has code to make sure it is doing only that.

Accordingly wrapr::let() treats both names/symbols and strings as symbols.

# Let substitution treats all substitutions as source-text

# so strings and names are as if the user had typed them

# in and behave as names (becoming the name of a column).

let(c(COL\_STRING = col\_string),

cat(to\_sql(d\_rep %.>% select\_rows(., is.na(COL\_STRING)),

db\_info)))

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_90437192531349864130\_0000000000

## WHERE ( ( \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_ ) IS NULL )

# Let substitution treats all substitutions as source-text

# so strings and names are as if the user had typed them

# in and behave as names (becoming the name of a column).

let(c(COL\_NAME = col\_name),

cat(to\_sql(d\_rep %.>% select\_rows(., is.na(COL\_NAME)),

db\_info)))

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_97451251523316128028\_0000000000

## WHERE ( ( \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_ ) IS NULL )

wrapr::let()‘s operating assumption is: if the user was using wrapr::let() the user was intending a symbol, regardless if they specify that symbol using a string or a symbol type. This means the user doesn’t have to maintain the distinction between string representations of names and symbol representations of names when using wrapr::let(). And again, for substituting string-values in: there are already much better ways, such as R evaluation itself (as we show below).

value\_we\_want <- "a"

let(c(COL\_NAME = col\_name),

cat(to\_sql(d\_rep %.>% select\_rows(., COL\_NAME == value\_we\_want),

db\_info)))

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_39226801511562936942\_0000000000

## WHERE \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_ = \_\_STRING\_CONSTANT\_\_a\_\_STRING\_CONSTANT\_\_

By assuming more about user intent wrapr::let() can smooth over inessential differences for the user.

**base::bquote() substitution**

bquote() substitution on the other hand is designed to substitute arbitrary values into un-executed language objects. This is the usual general definition of quasi-quotation, and is an emergent behavior. That we see the behavior one would expect by simply composing existing R language features. bquote() is what you get when you write reasonable code and then accept the resulting behavior as reasonable (even if the resulting behavior may or may not have been your first choice). This is in fact also a good design principle.

In this case the emergent behavior is: strings are treated as string constants, and names/symbols are treated as column names. That is the consequences of the substitution performed by bquote() is a function of the type of what is being substituted in. This actually makes sense, but it is something the user has to learn.

rquery can use bquote() substitution two ways: through its own NSE methods, or through wrapr:qe() (wrapr quote expression). Both work the same: they treat names/symbols as column names, and character/strings as string constants. So users must express their intent by passing in the correct type.

Here are examples to show the differences. In all cases substitution is triggered by the .()-notation.

# bquote substitution on string type: col\_string

# is taken to represent a string constant, not

# the name of a column.

cat(to\_sql(d\_rep %.>% select\_rows(., is.na(.(col\_string))),

db\_info))

## Warning in warn\_about\_filter\_conditions(parsed): rquery::select\_rows:

## expression is.na("x") refers to no columns (so is a constant)

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_48031370941868950818\_0000000000

## WHERE ( ( \_\_STRING\_CONSTANT\_\_x\_\_STRING\_CONSTANT\_\_ ) IS NULL )

# bquote substitution on name type: col\_name

# is taken to represent a column name.

cat(to\_sql(d\_rep %.>% select\_rows(., is.na(.(col\_name))),

db\_info))

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_31583525690015673048\_0000000000

## WHERE ( ( \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_ ) IS NULL )

# bquote substitution on string type: col\_string

# is taken to represent a string constant, not

# the name of a column.

cat(to\_sql(d\_rep %.>% select\_rows\_se(., qe(is.na(.(col\_string)))),

db\_info))

## Warning in warn\_about\_filter\_conditions(parsed): rquery::select\_rows:

## expression is.na("x") refers to no columns (so is a constant)

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_04319820330673360614\_0000000000

## WHERE ( ( \_\_STRING\_CONSTANT\_\_x\_\_STRING\_CONSTANT\_\_ ) IS NULL )

# bquote substitution on name type: col\_name

# is taken to represent a column name.

cat(to\_sql(d\_rep %.>% select\_rows\_se(., qe(is.na(.(col\_name)))),

db\_info))

## SELECT \* FROM (

## SELECT

## \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_

## FROM

## \_\_IDENTIFIER\_\_d\_\_IDENTIFIER\_\_

## ) tsql\_61704616114459293739\_0000000000

## WHERE ( ( \_\_IDENTIFIER\_\_x\_\_IDENTIFIER\_\_ ) IS NULL )

**Conclusion**

wrapr::let() behavior is an example of a forced design: a desirable effect is identified (in this case the ability to substitute in names from variables) and the implementation guarantees this effect. Because the implementation is attempting complete control of semantics we can precisely determine user visible effects. We can bend the implementation to our teaching. wrapr::let() is working *around* the R language, but deliberately doing so in a very narrow way (we are not re-implementing all of the evaluation path!).

base::bquote() behavior is an example of an emergent design: the code that is natural to get the desired functionality is written, and the exact consequences and details of the implementation are derived from the underlying language semantics. Because the implementation is not trying to work *around* the underlying language the semantics tend to be good and compatible with other parts of the language.

Both strategies are valid and have their advantages. I feel this in contrast to systems that re-implement very many (or even every) step of expression representation and evaluation. Once one overrides and re-implements all aspects of representation and evaluation one has two incompatible languages (the original and the overridden) bolted together to great confusion.