R documentation

of ‘ast2ast/man/translate.Rd’

September 15, 2022

`translate` *Translates an R function into a C++ function.*

Description

An R function is translated to C++ source code and afterwards the code is compiled.
The result can be an external pointer (*XPtr*) or an *R* function.
The default value is an R function.
Further information can be found in the vignette: *Detailed Documentation*.

Usage

```
translate(
  f,
  output = "R",
  types_of_args = "SEXP",
  return_type = "SEXP",
  reference = FALSE,
  verbose = FALSE
)
```

Arguments

`f`	The function which should be translated from R to C++.
`output`	If set to "R"" an R function wrapping the C++ code is returned. If output is set to "XPtr"" an external pointer object pointing to the C++ code is returned. The default value is "R".

types_of_args	define the types of the arguments passed to the function as an character vector. This is an optional input if using "XPtr"" as output. The default value is "SEXP"" as this is the only possibility for output "R". In case one want to use an external pointer the easiest way is to pass "sexp"" for types_of_args. Beyond that it is possible to pass "ptr_vec" and "ptr_mat". For more information see below for details and check the vignette *InformationForPackageAuthors*. **Beyond that, be aware that the passed *SEXP* objects are only copied if the object size increases. Thus, R objects can be modified within the function! See in section details for an example**
return_type	is a character defining the type which the function returns. The default value is "SEXP"" as this is the only possibility for output "R". Additionally, the possibilities "sexp"" and "void"" exist for the external pointer interface.
reference	If set to TRUE the arguments are passed by reference (not possible if output is "R").
verbose	If set to TRUE the output of the compilation process is printed.

Details

The types *numeric vector* and *numeric matrix* are supported. Notably, it is possible that the variables change the type within the function.
Beyond that, be aware that the passed *SEXP* objects are only copied if the size increases. Thus, R objects can be modified within the function!
For example in the following code the variable *a* contains *1, 2, and 3* before the function call and afterwards *1, 1 and 1*. In contrast for variable *b* the size changes and thus the object within *R* is not modified. Furthermore, the variable *c* is not increased and only the first element is changed.

```
f <- function(a, b, c) {
  a[c(1, 2, 3)] <- 1
  b <- vector(10)
  c <- vector(1)
}
fcpp <- ast2ast::translate(f)
a <- c(1, 2, 3)
b <- c(1, 2, 3)
c <- c(1, 2, 3)
fcpp(a, b,c)
print(a)
print(b)
print(c)
```

It is possible to declare a variable of a scalar numeric data type. This is done by adding *_db* to the end of the variable. Each time *_db* is found the variable is declared as a scalar numeric data type. In this case the object cannot change its type! In the example below the variable *a_db* is of type double whereas *b* is of type "sexp".

```
f <- function() {
  a_db = 3.14
  b = 3.14
}
fcpp <- ast2ast::translate(f, verbose = TRUE)
fcpp()
```

In R every object is under the hood a *SEXP* object. In case an *R* function is created as output only *SEXP* elements can be passed to the function. Furthermore, these functions always return a *SEXP* element. Even if nothing is returned; in this case *NULL* is returned!. Notably, is that only numeric vectors (in R also scalar values are vectors) or numeric matrices can be passed to the function.
In contrast if an external pointer is created other types can be specified which are passed to the function or returned from it. The default value is a variable of type *sexp*. This is the data type which is used in the C++ code. The *ptr_vec* and *ptr_mat* interface work in a different way. If using *ptr_vec* a *double** pointer is expected as first element. Additionally a second argument is needed which is of type *int* and which defines the size of the array. This works in the same way for *ptr_mat*. But instead of the size argument two integers are needed which define the number of rows and columns. Both arguments have to be of type *int*. Notably, the memory is only borrowed. Thus, the memory is not automatically deleted! See vignette *InformationForPackageAuthors* for more information.
The following functions are supported:

1. assignment: = and <-
2. allocation: vector and matrix
3. information about objects: length and dim
4. Basic operations: +, -, *, /
5. Indices: *'[]'* and at
6. mathematical functions: sin, asin, sinh, cos, acos, cosh, tan, atan, tanh, sqrt, log, ^ and exp
7. concatenate objects: c
8. control flow: for, if, else if, else
9. comparison: ==, !=, >, <, >= and <=
10. printing: print
11. returning objects: return
12. catmull-rome spline: cmr
13. to get a range of numbers the ':' function can be used
14. is.na and is.infinite can be used to test for NA and Inf.
15. d-, p-, q- and r-unif, -norm, -lnorm and -gamma (for gamma argument *Scale* cannot be defined and is calculated using *1/rate*)

Some details about the implemented functions

- allocation of memory works: Following forms are possible:
  - vector(size_of_elements)
  - vector(value, size_of_elements)
  - matrix(nrows, ncols)

  - matrix(value, nrows, ncols)
  - matrix(vector, nrows, ncols)
- For indices squared brackets *'[]'* can be used as common in R. Beyond that the function 'at' exists which accepts as first argument a variable and as the second argument you pass the desired index. The caveat of using 'at' is that only **one** entry can be accessed. The function *'[]'* can return more then one element.
  The *at*-function returns a reference to the vector entry. Therefore variable[index] can behave differently then *at(variable, index)*. The function has to be used carefully when *at* is used. Especially if *'[]'* and *at* are mixed the function behaviour is difficult to predict. Please test it before using it in a serious project.
  Here is a small example presented how to use the subset functions:

```
f <- function() {
  a <- c(1, 2, 3)
  print(at(a, 1))
  print(a[1:2])
}
fcpp <- ast2ast::translate(f)
fcpp()
```

- For-loops can be written as common in R
  - Nr.1
    for(index in variable){
    # do whatever
    }
  - Nr.2
    for(index in 1:length(variable){
    # do whatever
    }
- Be aware that it is possible to assign the result of a comparison to a variable. Example see below:
  However, the vector will contain only *0* or *1* instead of *FALSE* or *TRUE*.

```
a = c(1, 2, 3)
b = c(1, 2, 1)
c = a != b
```

- The print function accepts either a scalar, vector, matrix, string, bool or nothing (empty line).
- In order to return an object use the *return* function (**The last object is not returned automatically as in R**).
- In order to interpolate values the *cmr* function can be used. The function needs three arguments.
  1. the first argument is the point of the independent variable (**x**) for which the dependent variable should be calculated (**y**). This has to be a vector of length one.
  2. the second argument is a vector defining the points of the independent variable (**x**). This has to be a vector of at least length four.

3. the third argument is a vector defining the points of the dependent variable (**y**). This has to be a vector of at least length four.

Be aware that the R code is translated to ETR an expression template library which tries to mimic R.
However, it does not behave exactly like R! Please check your compiled function before using it in a serious project. If you want to see how *ast2ast* differs from R in detail check the vignette: *Detailed Documentation*.

Value

If output is set to *R* an R function is returned. Thus, the C++ code can directly be called within R. In contrast a function which returns an external pointer is generated if the output is set to *XPtr*.

Examples

```
# Further examples can be found in the vignettes.
   ## Not run:
     # Hello World
     # ============================================================================

     # Translating to R_fct
     # ----------------------------------------------------------------------------
     f <- function() { print("Hello World!")}
     ast2ast::translate(f)
     f()

     # Translating to external pointer
     # ----------------------------------------------------------------------------
     f <- function() { print("Hello World!")}
     pointer_to_f_cpp <- ast2ast::translate(f,
                                            output = "XPtr", return_type = "void")
     Rcpp::sourceCpp(code = '
     #include <Rcpp.h>
     typedef void (*fp)();

     // [[Rcpp::export]]
     void call_fct(Rcpp::XPtr<fp> inp) {
       fp f = *inp;
       f(); } ')

     call_fct(pointer_to_f_cpp)

     # Run sum example:
     # ============================================================================

     # R version of run sum
     # ----------------------------------------------------------------------------
     run_sum <- function(x, n) {
       sz <- length(x)

       ov <- vector(mode = "numeric", length = sz)
```

```
  ov[n] <- sum(x[1:n])
  for(i in (n+1):sz) {

    ov[i] <- ov[i-1] + x[i] - x[i-n]
  }

  ov[1:(n-1)] <- NA

  return(ov)
}

# translated Version of R function
# ------------------------------------------------------------------------------
run_sum_fast <- function(x, n) {
  sz <- length(x)
  ov <- vector(sz)

  sum_db = 0
  for(i in 1:n) {
    sum_db <- sum_db + at(x, i)
  }
  ov[n] <- sum_db

  for(i in (n + 1):sz) {
    ov[i] <- at(ov, i - 1) + at(x, i) - at(x, i - at(n, 1))
  }

  ov[1:(n - 1)] <- NA

  return(ov)
}
run_sum_cpp <- ast2ast::translate(run_sum_fast, verbose = FALSE)
set.seed(42)
x <- rnorm(10000)
n <- 500
one <- run_sum(x, n)
two <- run_sum_cpp(x, n)

## End(Not run)
```

Index