Information for package authors

Konrad Krämer

Guidance for Package Authors

This section of the documentation describes how the external pointers to C++ function that are produced by *ast2ast* can be used in other packages. This information is intended for package authors who want to use the translated functions within their own code. Additionally, this section endeavors to present pertinent implementation details, providing valuable insights into the inner workings of the process.

Naming rationalization ast2ast

The nomenclature of the package, ast2ast, is rooted in the abbreviation *ast*, signifying abstract syntax tree. Originally I planned to convert the abstract syntax tree of R to the C++ tree as the literature recommended for transpilers. However, through iterative refinement, a more optimal methodology emerged. The development incorporated an Expression Template Library in C++ known as ETR, meticulously crafted to mimic R. Thus, R code is translated into ETR code, which is subsequently compiled. The original ETR library is accessible at https://github.com/Konrad1991/ETR. It's imperative to note that the version integrated into ast2ast has undergone substantial enhancements, amplifying its efficacy and adaptability.

Comparison of R and ETR code

contents…

contents…

Variables

The core class object is named Vec<T, R, Trait>. Here the template *T* represents a basic data type. The and can be initialized with any type. Furthermore, is a typedef defined for VEC called *sexp* alluding to the fact that all R objects are *SEXP* objects at C level. This class contains another class called STORE which manages memory. To do this a raw pointer of the form *T** is used. Thus, all objects are located at the heap. Beyond that, it is important that no static methods are implemented or that memory is associated with functions or global variables. Therefore, the VEC objects can be used in parallel. However, the user has to take care that only one thread edits an object at a time. VEC a.k.a. *sexp* can be converted to Rcpp::NumericVectors, Rcpp::NumericMatrices, arma::vec and arma::mat or to SEXP. Moreover, it is also possible to copy the data from Rcpp::NumericVectors, Rcpp::NumericMatrices, arma::vec or arma::mat to a *sexp* variable. The constructors and the *operator=* are implemented for this conversions. It is possible to construct a *sexp* object by using a *double** pointer and a size information (= int). The information about the Rcpp-, RcppArmadillo- and pointer-interface is explained in depth in the sections below.

If the user creates an R function all input arguments are of type *SEXP*. Furthermore, is the output always of type *SEXP*. Beyond that, there exists the possibility to create a function returning an external pointer to the C++ function. If using this interface the parameter passed to the function can be of one of the following types as long as an *sexp* object is defined as argument for the function.

- double
- SEXP
- sexp
- Rcpp::NumericVector or NumericVector
- Rcpp::NumericMatrix or NumericMatrix
- arma::vec or vec
- arma::mat or mat

The variables are directly converted to sexp thereby copying the memory. Thus, the variable can not be used if an function expects a reference of type *sexp* If the aim is to modify a variable by the user function it is only possible to pass directly a *sexp* objects by reference. See the example below here a is modified whereas b is copied and thus not altered. The vector *v* can only be used for the second argument of the function *foo*.

```
// [[Rcpp::plugins(cpp20)]]
// [[Rcpp::depends(RcppArmadillo, ast2ast)]]
#include "etr.hpp"

void foo(sexp& a, sexp b) {
  a = b + 1.0;
  b = 1.0;
}

// [[Rcpp::export]]
void call_fct() {
  sexp a = etr::coca(1, 2, 3);
  sexp b = etr::coca(1, 2, 3);
  foo(a, b);
  etr::print(a);
  etr::print(b);
}
```

```
call_fct()
```

```
## 2 3 4
## 1 2 3
```

Beyond that using "XPtr" as *output* argument of the function translate it is possible to specify **ptr_vec** or a **ptr_mat** as desired types. If these types are used sexp objects are created which form a shallow wrapper around these pointers. See an example below how this works.

1. a Rcpp function is created which calls the translated R code:

```
// [[Rcpp::depends(ast2ast)]]
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::plugins(cpp20)]]
#include "etr.hpp"

#include <Rcpp.h>
using namespace Rcpp;

typedef sexp (*fct_ptr) (double* a_double_ptr, int a_int_size);
```

```
// [[Rcpp::export]]
Rcpp::NumericVector fcpp(XPtr<fct_ptr> f) {
  fct_ptr fct = *f;

  int size = 10;
  double* ptr = new double[size];
  for(int i = 0; i < size; i++) {
    ptr[i] = static_cast<double>(i);
  }

  Rcpp::NumericVector ret = fct(ptr, size);

  delete[] ptr;

  return ret;
}
```

2. a R function is defined and translated (not shown):

```
f <- function(a) {
  d_db = 1
  ret <- a + 2 + d_db
  return(ret)
}
```

3. The translated R function has the following code:

```
SEXP f(double* a_double_ptr, int a_int_size ) {

double d_db;
sexp ret;
sexp a(a_int_size, a_double_ptr, 2);
d_db = etr::i2d(1);
ret = a + etr::i2d(2) + d_db;
return(ret);
}
```

How to use ast2ast

In this paragraph, a basic example demonstrates how to write the R code, translate it and call it from C++. Particular emphasis is placed on the C++ code. First of all, the R function is defined which accepts one argument called *a*, adds two to *a* and stores it into *b*. The variable *b* is returned at the end of the function. The R function called *f* is translated to an external pointer to the *C++* function.

```
f <- function(a) {
  b <- a + 2
  return(b)
}
library(ast2ast)
f_cpp <- translate(f, output = "XPtr", types_of_args = "sexp", return_type = "sexp")
```

The C++ function depends on RcppArmadillo and ast2ast therefore the required macros and headers were included. Moreover, *ETR* requires std=c++17 therefore the corresponding plugin is added. The function *getXPtr* is defined and is the function which is returned. In the last 5 lines, the translated code is depicted. The function *f* returns a *sexp* and gets one argument of type *sexp* called *a*. The body of the function looks almost identical to the R function. Except that the variable *b* is defined in the first line of the body with the type *sexp*. The function *i2d* converts an integer to a double variable. This is necessary since C++ would identify the *2* as an integer which is not what the user wants in this case.

```
// [[Rcpp::depends(ast2ast, RcppArmadillo)]]
// [[Rcpp::plugins(cpp20)]]
#include <etr.hpp>

// [[Rcpp::export]]
SEXP getXPtr();

sexp f(sexp a) {
  sexp b;
  b = a + etr::i2d(2);
  return(b);
}
```

Afterwards, the translated R function has to be used in C++ code. This would be your package code for example. First, the macros were defined for *RcppArmadillo* and *ast2ast*. Subsequently, the necessary header files were included. As already mentioned *ast2ast* requires std=c++17 thus the required plugin is included. To use the function, it is necessary to dereference the pointer. The result of the dereferenced pointer has to be stored in a function pointer. Later the function pointer can be used to call the translated function. Therefore, a function pointer called *fp* is defined. It is critical that the signature of the function pointer matches the one of the translated function. Perhaps it would be a good idea to check the R function before it is translated. After defining the function pointer, a function is defined which is called later by the user (called *call_package*). This function accepts the external pointer. Within the function body, a variable *f* is defined of type *fp* and *inp* is assigned to it. Next, a *sexp* object called *a* is defined which stores a vector of length 3 containing 1, 2 and 3. The function *coca* is equivalent to the *c* function R. Afterwards *a* is printed. Followed by the call of the function *f* and storing the result in *a*. The variable *a* is printed again to show that the values are changed according to the code defined in the R function.

```
// [[Rcpp::depends(RcppArmadillo, ast2ast)]]
#include "etr.hpp"
// [[Rcpp::plugins("cpp20")]]
typedef sexp (*fp)(sexp a);
using namespace etr;

// [[Rcpp::export]]
void call_package(Rcpp::XPtr<fp> inp) {
  fp f = *inp;
  sexp a = coca(1, 2, 3);
  print(a);
  a = f(a);
  print("a is now:");
  print(a);
}
```

The user can call now the package code and pass the R function to it. Thus, the user only has to install the compiler or Rtools depending on the operating system. But it is not necessary to write the function in Rcpp.

```
call_package(f_cpp)
```

```
## 1 2 3
## a is now:
##
## 3 4 5
```

Rcpp Interface

In the last section, the usage of *ast2ast* was described. However, only *sexp* variables were defined. Which are most likely not used in your package. Therefore interfaces to common libraries are defined. First of all, *ast2ast* can communicate with Rcpp which alleviates working with the library substantially. The code below shows that it is possible to pass a *sexp* object to a variable of type *NumericVector* or *NumericMatrix* and *vice versa*. Here, the data is always copied.

```
// [[Rcpp::depends(ast2ast, RcppArmadillo)]]
#include <RcppArmadillo.h>
#include <Rcpp.h>
// [[Rcpp::plugins(cpp20)]]
using namespace Rcpp;
#include <etr.hpp>
using namespace etr;


// [[Rcpp::export]]
void fct() {
  // NumericVector to sexp
  NumericVector a{1, 2};
  sexp a_ = a;
  print(a_);

  // sexp to NumericVector
  sexp b_ = coca(3, 4);
  NumericVector b = b_;
  Rcpp::Rcout << b << std::endl;

  // NumericMatrix to sexp
  NumericMatrix c(3, 3);
  sexp c_ = c;
  print(c_);

  // sexp to NumericMatrix
  sexp d_ = matrix(colon(1, 16), 4, 4);
  NumericMatrix d = d_;
  Rcpp::Rcout << d << std::endl;
}
```

```
trash <- fct()
```

```
## 1 2
## 3 4
## 0    0   0
## 0    0   0
## 0    0   0
## 1.00000 5.00000 9.00000 13.0000
## 2.00000 6.00000 10.0000 14.0000
## 3.00000 7.00000 11.0000 15.0000
## 4.00000 8.00000 12.0000 16.0000
```

RcppArmadillo Interface

Besides Rcpp types, *sexp* objects can transfer data to *RcppArmadillo* objects and it is also possible to copy the data from *RcppArmadillo* types to *sexp* objects using the operator =. The code below shows that it is possible to pass a *sexp* object to a variable of type *vec* or *mat* and *vice versa*. Here the data is always copied.

```
// [[Rcpp::depends(ast2ast, RcppArmadillo)]]
#include <RcppArmadillo.h>
#include <Rcpp.h>
// [[Rcpp::plugins(cpp20)]]
using namespace arma;
#include <etr.hpp>
using namespace etr;

// [[Rcpp::export]]
void fct() {
  // vec to sexp
  arma::vec a(4, fill::value(30.0));
  sexp a_ = a;
  print(a_);

  // sexp to vec
  sexp b_ = coca(3, 4);
  vec b = b_;
  b.print();

  // mat to sexp
  mat c(3, 3, fill::value(31.0));
  sexp c_ = c;
  print(c_);

  // sexp to mat
  sexp d_ = matrix(colon(1, 16), 4, 4);
  mat d = d_;
  d.print();
}
```

```
trash <- fct()
```

```
## 30 30 30 30
##    3.0000
##    4.0000
## 31   31  31
## 31   31  31
## 31   31  31
##     1.0000    5.0000    9.0000   13.0000
##     2.0000    6.0000   10.0000   14.0000
##     3.0000    7.0000   11.0000   15.0000
##     4.0000    8.0000   12.0000   16.0000
```

Pointer Interface

You can pass the information of data stored on heap to a *sexp* object. The constructor for type vector accepts 3 arguments:

- *int* defining the size of the data.
- a pointer ($T*$) to the data
- *int* called cob (copy, ownership, borrow).

The constructor for the type matrix accepts 4 arguments:

- *int* defining number of rows
- *int* defining number of cols
- a pointer (*T**) to the data
- *int* called cob (copy, ownership, borrow).

If cob is 0 then the data is copied. Else if cob is 1 then the pointer itself is copied. Meaning that the ownership is transferred to the *sexp* object and the user should not call delete [] on the pointer. Be aware that only one *sexp* variable can take ownership of one vector otherwise the memory is double freed. Else if cob is 2 the ownership of the pointer is only borrowed. Meaning that the *sexp* object cannot be resized. The user is responsible for freeing the memory! The code below shows how the pointer interface works in general. Showing how *sexp* objects can be created by passing the information of pointers (*double**) which hold data on the heap. Currently, only constructors are written which can use the pointer interface.

```
// [[Rcpp::depends(ast2ast, RcppArmadillo)]]
#include <RcppArmadillo.h>
#include <Rcpp.h>
// [[Rcpp::plugins(cpp20]]
using namespace arma;
#include <etr.hpp>
using namespace etr;

// [[Rcpp::export]]
void fct() {
    int size = 3;

    // copy
    double* ptr1;
    ptr1 = new double[size];
    int cob = 0;
    sexp a(size, ptr1, cob);
    delete [] ptr1;
    a = vector(3.14, 5);
    print(a);

    print();

    // take ownership
    double* ptr2;
    ptr2 = new double[size];
    cob = 1;
    sexp b(size, ptr2, cob);
    b = vector(5, 3);
    print(b);

    print();

    // borrow ownership
    double* ptr3;
    ptr3 = new double[size];
    cob = 2;
    sexp c(size, ptr3, cob);
    //error calls resize
    //c = vector(5, size + 1);
    c = vector(4, size);
    print(c);
    print();
    sexp d(size, ptr3, cob);
    d = d + 10;
    print(d);
    print();

    delete[] ptr3;
}
```

```
trash <- fct()
```

```
## 30 30 30 30
##    3.0000
##    4.0000
## 31   31  31
## 31   31  31
## 31   31  31
##     1.0000    5.0000    9.0000   13.0000
##     2.0000    6.0000   10.0000   14.0000
##     3.0000    7.0000   11.0000   15.0000
##     4.0000    8.0000   12.0000   16.0000
```

The pointer interface is particularly useful if the user function has to change the data of a vector or matrix of type *NumericVector*, *vec*, *NumericMatrix* or *mat*. Assuming that the user passes a function that accepts its arguments by reference it is easy to modify any variable which has a type that can return a pointer to its data. In the code below it is shown how *sexp* objects are constructed using the pointer interface. Thereby changing the content of variables which has an Rcpp type, a RcppArmadillo type, or is of type std::vector.

```
// [[Rcpp::depends(ast2ast, RcppArmadillo)]]
#include <RcppArmadillo.h>
#include <Rcpp.h>
// [[Rcpp::plugins(cpp17)]]
using namespace Rcpp;
using namespace arma;
#include <etr.hpp>
using namespace etr;

typedef void (*fp)(sexp& a);

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

  // NumericVector
  NumericVector a_rcpp{1, 2, 3};
  sexp a(a_rcpp.size(), a_rcpp.begin(), 2);
  f(a);
  Rcpp::Rcout << a_rcpp << std::endl;

  //arma::vec
  vec a_arma(2, fill::value(30));
  sexp b(2, a_arma.memptr(), 2);
  f(b);
  a_arma.print();
  // NumericMatrix
  NumericMatrix c_rcpp(2, 2);
  sexp c(2, 2, c_rcpp.begin(), 2);
  f(c);
  Rcpp::Rcout << c_rcpp << std::endl;

  //arma::mat
  mat d_arma(3, 2, fill::value(30));
  sexp d(3, 2, d_arma.memptr(), 2);
  f(d);
  d_arma.print();
}
```

```
f <- function(a) {
  a <- a + 2
}

library(ast2ast)
fa2a <- translate(f, reference = TRUE, output = "XPtr", types_of_args = "sexp", return_type = "void")
trash <- call_package(fa2a)
```