Package: ast2ast (via r-universe)

October 30, 2026

Type Package

Title Translates an R Function to a C++ Function

Version 1.1.2

Date 2026-10-28

Maintainer Krämer Konrad <konrad_kraemer@yahoo.de>

BugReports https://github.com/Konrad1991/ast2ast/issues

URL https://github.com/Konrad1991/ast2ast

Description Enable translation of a tiny subset of R to C++. The user has to define a R function which gets translated. For a full list of possible functions check the documentation. After translation an R function is returned which is a shallow wrapper around the C++ code. Alternatively an external pointer to the C++ function is returned to the user. The intention of the package is to generate fast functions which can be used as ode-system or during optimization.

License GPL (>= 2)

Imports Rcpp (>= 1.0.4), R6, methods, RcppThread, utils, sundials

LinkingTo Rcpp, RcppThread

Depends R (>= 4.1.0)

SystemRequirements GNU make, C++20, Rtools (>= 4.5) on Windows

NeedsCompilation yes

VignetteBuilder knitr

Suggests knitr, kableExtra, rmarkdown, tinytest, microbenchmark, ggplot2, RcppXPtrUtils, pkgbuild, nnls, Matrix

Encoding UTF-8

Author Krämer Konrad [aut, cre]

Config/pak/sysreqs make

Repository https://cran.r-universe.dev

Date/Publication 2026-10-30 19:32:44 UTC

RemoteUrl https://github.com/cran/ast2ast

RemoteRef HEAD

RemoteSha bc58088e011a9cdc2800c8a3a2c1949ac099da1c

Contents

make_package 2
translate 4

Index **13**

`make_package` *Translate R files into the C++ code of an existing package*

Description

Works like `Rcpp::compileAttributes`: it extends an existing package. Each entry point is translated into its own C++ file, together with the helpers it uses; only the entry points can be called from R.

Usage

```
make_package(files, entry_points, package_dir = ".", overwrite = FALSE,
             derivative = list(), debug = TRUE)
```

Arguments

`files`	Character vector of R files, relative to `package_dir`. The files must not be located in `R/`, since R sources everything in `R/` when the package is installed. `inst/tools/` is a good place for them.
`entry_points`	Names of the functions that are exported to R. All other functions in `files` are internal helpers. A helper that no entry point uses (directly or via other helpers) raises a warning.
`package_dir`	Directory of the package (the one containing the `DESCRIPTION` file). Defaults to the current directory.
`overwrite`	Logical. Files that were not generated by `make_package` are only overwritten after confirmation. In a non-interactive session an error is raised unless `overwrite = TRUE`.
`derivative`	Named list: the mode of the automatic differentiation per entry point, `"forward"` or `"reverse"` (see `translate`). Entry points not listed need no derivatives. E.g. `list(g = "reverse")`.
`debug`	Logical, as in `translate`: `TRUE` reports the failing statement on runtime errors. `FALSE` is faster.

Details

The files may only contain two kinds of top-level code:

- type definitions: `new_type(...)`, see `translate`.
- function definitions: `name <- function(...) { ... }`. Every function starts with `argtypes(...)` and `returntype(...)`.

Everything else (e.g. `x <- 1` or `library(...)`) is an error.

The following files are written:

- `src/ast2ast_<entry point>.cpp`: the translated code, one file per entry point. Thus, each entry point can use its own `derivative`. Generated files of entry points that no longer exist are removed.
- `src/Makevars` and `src/Makevars.win`.
- `inst/include/ast2ast/`: the headers of ast2ast (GPL (>= 2)).
- `R/RcppExports.R` and `src/RcppExports.cpp` via `Rcpp::compileAttributes`.

Generated files start with the line `Generated by ast2ast::make_package -- do not edit` and are overwritten without asking. After changing the R files, call `make_package` again.

The package has to be set up beforehand. `make_package` only checks this and reports what is missing:

- `DESCRIPTION`: `Imports: Rcpp` and `LinkingTo: Rcpp, RcppThread, sundials`.
- `NAMESPACE`: `useDynLib(<package>, .registration = TRUE)` and `importFrom(Rcpp, evalCpp)`.

Entry points are not exported automatically; add `export(...)` to the `NAMESPACE` if users should call them.

Runtime errors name the package, the file, the line and the function, e.g. `In 'mypkg: helpers.R:5 (helper): return(v[[i]])': ....`

Value

The package name, invisibly.

See Also

`translate`

Examples

```
dir <- file.path(tempdir(), "mypkg")
dir.create(file.path(dir, "inst", "tools"), recursive = TRUE)
writeLines(c("Package: mypkg", "Version: 0.0.1", "Title: Example",
             "Imports: Rcpp", "LinkingTo: Rcpp, RcppThread, sundials"),
           file.path(dir, "DESCRIPTION"))
writeLines(c("useDynLib(mypkg, .registration = TRUE)",
             "importFrom(Rcpp, evalCpp)", "export(f)"),
           file.path(dir, "NAMESPACE"))
writeLines(c(
```

```
  "f <- function(a) {",
  "  argtypes(a |> type(double))",
  "  returntype(double)",
  "  b <- 2.0",
  "  helper(b) * a",
  "}",
  "helper <- function(b) {",
  "  argtypes(b |> type(double))",
  "  returntype(double)",
  "  b + 1",
  "}"
), file.path(dir, "inst", "tools", "f.R"))
ast2ast::make_package("inst/tools/f.R", "f", dir)
list.files(file.path(dir, "src"))
```

`translate` *Translate an R function into C++*

Description

`translate()` compiles an R function into C++ code using the *ast2ast* expression template library. The result can be called directly from R (default) or returned as an external pointer. The C++ source code can also be retrieved without compilation.

Usage

```
translate(
  f,
  types_f = NULL,
  output = "R",
  derivative = NULL,
  verbose = FALSE,
  getsource = FALSE,
  debug = TRUE
)
```

Arguments

`f`	An R function to be translated into C++. Argument types are declared with an `argtypes(...)` block as the first statement of its body, and the return type optionally with `returntype(...)` right after it; see the Details section.
`types_f`	A helper function that declares custom struct types via `new_type()`, for use as argument types, variable types, or slot types inside `f`. Optional; defaults to `NULL`. See the Details section.
`output`	Controls what is returned: • `"R"` (default): an R function wrapping the compiled C++.

- `"XPtr"`: an external pointer to the compiled C++ function.

`derivative` When derivatives are required one can set here the mode of the automatic differentiation:

- `NULL` (default): You don't require any derivatives.
- `"forward"` Determine the derivatives via forward automatic differentiation.
- `"reverse"` Determine the derivatives via reverse automatic differentiation.

`verbose` Logical. If `TRUE`, prints output from the compilation process.

`getsource` Logical. If `TRUE`, returns the generated C++ source code as a string instead of compiling it.

`debug` Logical. Defaults to `TRUE`. When `TRUE`, a runtime error is reported as `"In '<statement>': <message>"`, naming the specific line of translated code that failed. Set to `FALSE` to get the plain, unattributed message instead.

Details

Supported functions: The following R constructs are currently supported:

1. Assignment: `=`, `<-`, and `$` (struct field access/assignment)
2. Allocation: `vector`, `matrix`, `array`, `rep`, `logical`, `integer`, `numeric`
3. Object info: `length`, `dim`, `nrow`, `ncol`
4. Arithmetic: `+`, `-`, `*`, `/`, `%%`, `%/%`, `%*%`, `^`
5. Indexing: `[]`, `[[]]`, and `at()` (an alias for `[[]]`)
6. Math: `sin`, `asin`, `sinh`, `cos`, `acos`, `cosh`, `tan`, `atan`, `tanh`, `sqrt`, `abs`, `sign`, `log`, `exp`, `floor`, `ceiling`, `trunc`, `round`
7. Reductions: `sum`, `prod`, `mean`, `max`, `min`, `which.max`, `which.min`, `which`, `all`, `any`
8. Vector transforms: `rev`, `sort`, `cumsum`
9. Matrix margins: `colSums`, `rowSums`, `colMeans`, `rowMeans`
10. Selection: `ifelse`
11. Linear algebra: `t`, `chol`, `crossprod`, `tcrossprod`, `diag`, `get_diag`, `solve`, `backsolve`, `forwardsolve`, `rbind`, `cbind`
12. Root finding / least squares: `uniroot`, `nnls`
13. Functionals (take an `fn()`): `map`, `Reduce`, `Filter`, `apply`
14. Optimizers / derivatives of an `fn()`: `jacobian`, `jacobian_sparse`, `jacobian_banded`, `lbfgsb`, `pso`
15. Casts: `as.numeric`, `as.integer`, `as.logical`
16. Sequences: `seq_len`, `seq_along`
17. Concatenation: `c`
18. Control flow: `for`, `while`, `repeat`, `next`, `break`, `if`, `else if`, `else`
19. Comparison: `==`, `!=`, `>`, `<`, `>=`, `<=`
20. Logical ops: `&&`, `||`, `&`, `|`, `!`
21. Printing: `print`
22. Errors: `stop`
23. Return: `return`
24. Catmull–Rom spline: `cmr`

25. Sequence operator: `:`
26. Helpers: `is.na`, `is.nan`, `is.finite`, `is.infinite`
27. Explicit typing: `type()`
28. Derivative functions in forward mode: `seed`, `unseed`, and `get_dot`
29. Derivative functions in reverse mode: `deriv`
30. Inner functions: `fn()`, see below
31. Custom types: `new_type()`, see below
32. Layouts (names): `new_layout()`, `new_mat_layout()`, `new_array_layout()`, `new_collection_layout()`, see below

Types are static in C++ and cannot be changed within the function. Each type consists of a base data type and a data structure:

- Base types: `logical`, `integer` (or `int`), `double`
- Structures: `scalar`, `vector` (or `vec`), `matrix` (or `mat`)

Types are usually inferred automatically. Users may annotate explicitly, for example:

```
a |> type(logical)      # scalar logical
b |> type(vec(int))     # integer vector
c |> type(mat(double))  # double matrix
```

Scalars in ast2ast differ from R: in R, scalars are length-1 vectors, but in C++ they are true scalars and cannot be subset.

Argument types: Arguments default to `matrix(double)` if `f`'s body has no `argtypes(...)` block. To override, make `argtypes(...)` the first statement of the body, one `type()` annotation per argument:

```
f <- function(a, b, c) {
  argtypes(
    a |> type(borrow_vec(double)) |> ref(),
    b |> type(borrow_mat(double)) |> ref() |> const(),
    c |> type(double) |> ref()
  )
  # ... body ...
}
```

Supported extensions for arguments:

- `borrow_vec`, `borrow_mat`, `borrow_array`: pass inputs by reference, modifying them in place. Passed to an inner function expecting `vec`/`mat`/`array` they are copied (with a warning); a mutable `ref()` parameter requires the borrowed type.
- `const()`: disallow modification of inputs.
- `ref()`: pass by reference; changes persist after the call. Works for inner functions, and for the arguments of `f` only with `output = "XPtr"`.

Return type: The return type is normally inferred from `f`'s body. To declare it explicitly, add `returntype(...)` directly after `argtypes(...)` (or as the first statement if there is no `argtypes(...)`):

```
f <- function(x) {
  argtypes(x |> type(vec(double)))
  returntype(double)
  return(sum(x))
}
```

When given, every `return` in `f` is checked against the declared type instead of being inferred, for both `output = "R"` and `output = "XPtr"`. Use `returntype(void)` for a function with no return value.

Inner functions: Functions can be defined inside `f` with `fn()`, which takes three positional parts: `argtypes(...)` (argument types, same form as the outer `argtypes(...)` block), `return(...)` (the return type), and a `{ }` block (the function body; the braces are optional for a single statement). Inner functions may call each other, including recursively, and can be passed as values to functions expecting one, such as `uniroot()`. A non-`const()` parameter only binds to a bare variable at the call site, not to an arbitrary expression (e.g. `x + 1`); declare the parameter `const()` to accept expressions.

```
factorial <- fn(
  argtypes(a |> type(int) |> const()),
  return(int),
  {
    if (a == 1L) return(a) else return(a * factorial(a - 1L))
  }
)
```

`uniroot(f, interval, tol, maxiter)` takes a function of a single `double` returning a `double` (or two arguments if a fifth, extra-data argument is supplied to `uniroot`), and returns a struct with fields `root`, `f_root`, `iter`, and `estim_prec`. `nnls(A, b)` solves the non-negative least squares problem and returns the solution vector.

Functionals: These take an `fn()` as their first argument:

- `map(f, x, ...)`: apply `f` element-wise over the given vectors (scalars broadcast). The result shape follows `f`'s return type: scalar gives a vector, vector a matrix, matrix/array an array, `new_type` a `collection`.
- `pmap(f, ncores, x, ...)`: like `map`, but the elements are processed in parallel. `ncores` is a scalar int or double, clamped to the number of available cores. `f` may not call functions that use the R API internally (currently `lbfgsb`). Not available with `derivative = "forward"`/`"reverse"`.
- `Reduce(f, x)`: left fold seeded with `x[[1]]`; `f` is `fn(acc, elem) -> acc`.
- `Filter(f, x)`: keep the elements of vector `x` for which `f(elem)` is `TRUE`.
- `apply(f, MARGIN, x)`: function first (unlike base R). `x` is a matrix, `MARGIN` is 1 (rows) or 2 (columns), `f` maps a vector to a scalar or vector.

Optimizers and Jacobians:

- `jacobian(f, x[, data][, colouring])`: the `m`-by-`n` Jacobian at `x` of `f`, which maps a double vector to a double vector. Requires `derivative = "forward"` or `"reverse"`. The optional `colouring` (forward mode only) computes columns that share no row in one sweep:

  - `colouring = "static", n = <integer literal>`: the sparsity pattern of `f` is derived at translation time for `length(x) == n`.
  - `colouring = "runtime"`: the pattern is recorded at every call from one reverse-mode run of `f`; `f` has to be a named function.
- `jacobian_sparse(f, x[, data], colouring = ...)`: like `jacobian`, but only the entries of the sparsity pattern are computed and stored (CSC, 0-based, as SUNDIALS' `SUNMatrix_Sparse`). Fields: `$m`, `$n`, `$indexptrs`, `$indexvals`, `$data`.
- `jacobian_banded(f, x[, data], colouring = ...)`: square Jacobian stored as LAPACK band matrix (as SUNDIALS' `SUNMatrix_Band`): `A(i, j)` at `data[j * ldim + mu + i - j]` (0-based), `ldim = mu + ml + 1`. Fields: `$n`, `$mu`, `$ml`, `$ldim`, `$data`.

`jacobian_sparse` and `jacobian_banded` need the sparsity pattern, so `colouring` is required and only forward mode is supported. Both are returned to R as a `Matrix::dgCMatrix` (package **Matrix** has to be installed).

- `lbfgsb(f, x, lower, upper, maxit, factr, pgtol, lmm[, data])`: bound-constrained L-BFGS-B via R's own C routine. `f` maps a double vector to a scalar double. Under `derivative = "forward"/"reverse"` the gradient is exact (via `jacobian`); otherwise it is a central-difference approximation. Returns a struct with `$par`, `$value`, `$convergence`, `$counts`.
- `pso(f, lower, upper, ngen, npop, error_threshold, global, seed[, data])`: derivative-free particle-swarm optimisation. `f` maps a double vector to a scalar double; `seed` seeds pso's own Mersenne-Twister RNG; returns the best parameter vector found.

For `lbfgsb` and `pso` the optional trailing `data` argument (any non-function, non-character value) is passed to `f` unchanged as a second argument.

Custom types: User-defined struct types are declared in a `types_f` helper function (passed to `translate()` via `types_f =`) and created with `new_type()` and `slots()`:

```
types_f <- function() {
  new_type(Point, slots(x |> type(double), y |> type(double)))
}
```

On the R side, a value of a custom type is a named list with a matching `class` attribute, e.g. `structure(list(x = 1, y = 2), class = "Point")`. Fields are read and written with `$`, including chained access such as `s$circles[[1L]]$center$x`. A slot may itself be a custom type (nested structs) or `collection(TypeName)`, a vector of a custom type; allocate one locally with `vector(mode = "TypeName", n)`.

Named vectors and layouts: Layouts attach names to vectors, matrices, arrays and collections. They are declared in `types_f`; each names argument is an R expression, `NULL` (no names) or an existing `new_layout()`:

```
types_f <- function() {
  new_layout(State, double, names = c("prey", "pred"))
new_mat_layout(Result, double, rownames = c("t0", "t1"), colnames = State)
new_array_layout(Cube, double, dimnames = list(c("a", "b"), NULL, c("t1", "t2")))
  new_type(Point, slots(x |> type(double), y |> type(double)))
  new_collection_layout(Pts, Point, names = c("p1", "p2"))
}
```

Used as types `named_vec(State)` / `named_borrow_vec(State)`, `named_mat` / `named_borrow_mat`, `named_array` / `named_borrow_array` and `named_collection`. Name indices such as `y[["pred"]]` or `m[["t1", "pred"]]` (also mixed with numbers) are resolved to integer indices at translation time; a typo is an error with a suggestion. Names of R arguments are checked and returned objects carry their names. `vector("State")` or a declaration (`res |> type(named_vec(State))`) allocates one entry per name, `vector("State", n)` allocates `n` entries.

Derivatives: `ast2ast` supports automatic differentiation (AD) in two modes: forward and reverse. The mode is selected via the `derivative` argument of `translate()`.

- `derivative = "forward"` enables forward-mode AD.
- `derivative = "reverse"` enables reverse-mode AD.

The AD system is intentionally low-level and explicit: derivative computations are assembled from a small set of primitive building blocks (`seed`, `unseed`, `get_dot`, `deriv`), which keeps the interface transparent and close to the generated C++ code. A built-in `jacobian(f, x)` is also provided for the common case (see *Optimizers and Jacobians* above); it is itself implemented on top of those primitives.

Forward mode: In forward mode, derivatives are propagated alongside values. The following functions are available:

- `seed(x, i)`: marks the `i`-th component of `x` as the active direction (sets its derivative to 1).
- `unseed(x, i)`: resets the derivative state of the `i`-th component.
- `get_dot(y)`: extracts the derivative (dot) values of `y`.

A typical pattern is to loop over input dimensions, seed one component at a time, evaluate the function, extract derivatives using `get_dot()`, and assemble the Jacobian manually.

Reverse mode: In reverse mode, derivatives are accumulated by backpropagation from outputs to inputs. The function

- `deriv(y, x)`

computes the Jacobian of `y` with respect to `x`. This call must appear explicitly in the translated function. Reverse mode is particularly efficient when the number of outputs is small relative to the number of inputs.

Design philosophy: Derivative computation in `ast2ast` is explicit by design. The full control flow (including loops, seeding, unseeding, and accumulation) is visible in the user code and translated directly into C++. This makes the generated code easy to inspect, reason about, and modify, and avoids hidden performance costs. **Important:** The `derivative` argument only enables the AD infrastructure. It does not automatically differentiate your function. You must explicitly call `seed()`, `unseed()`, `get_dot()`, or `deriv()` inside your function, depending on the chosen mode.

Note: The generated C++ mimics R semantics closely but not exactly. Always validate compiled functions against the original R implementation before using in production. See the vignette *Detailed Documentation* for a full comparison, and *InformationForPackageAuthors* for internals.

Value

Depending on `output`:

- *R*: an R function that directly calls the compiled C++ code.
- *XPtr*: an external pointer (class `"XPtr"`) to the compiled C++ function.

Examples

```
## Not run:
# Hello World
# ------------------------------------------------------------------
f <- function() {
  print("Hello World!")
}
f_cpp <- ast2ast::translate(f)
f_cpp()

# Derivatives
# ------------------------------------------------------------------
f <- function(y, x) {
  y[[1L]] <- x[[1L]] * x[[2L]]
  y[[2L]] <- x[[1L]] + x[[2L]]*x[[2L]]
  jac <- deriv(y, x)
  return(jac)
}
fcpp_reverse <- ast2ast::translate(f, derivative = "reverse")
y <- c(0, 0)
x <- c(2, 3)
fcpp_reverse(y, x)

f <- function(y, x) {
  jac <- matrix(0.0, length(y), length(x))
  for (i in 1L:length(x)) {
    seed(x, i)
    y[[1L]] <- x[[1L]] * x[[2L]]
    y[[2L]] <- x[[1L]] + x[[2L]]*x[[2L]]
    d <- get_dot(y)
    jac[TRUE, i] <- d
    unseed(x, i)
  }
  return(jac)
}
fcpp_forward <- ast2ast::translate(f, derivative = "forward")
fcpp_forward(y, x)

# Bubble sort (using [[ for scalars)
# ------------------------------------------------------------------
bubble <- function(a) {
  size <- length(a)
  for (i in 1:size) {
    for (j in 1:(size - 1)) {
      if (a[[j]] > a[[j + 1]]) {
        temp <- a[[j]]
        a[[j]] <- a[[j + 1]]
        a[[j + 1]] <- temp
      }
    }
  }
  return(a)
```

```
}
bubble_cpp <- ast2ast::translate(bubble)
bubble_cpp(runif(10))

# Fibonacci sequence
# ------------------------------------------------------------------
fib <- function(n = 10) {
  f <- integer(n)
  f[[1L]] <- 1L
  f[[2L]] <- 1L
  for (i in 3L:n) {
    f[i] <- f[i-1L] + f[i-2L]
  }
  return(f)
}
fib_cpp <- ast2ast::translate(fib)
fib_cpp(10)

# Custom types
# ------------------------------------------------------------------
types_f <- function() {
  new_type(Point, slots(x |> type(double), y |> type(double)))
}
f <- function(p) {
  argtypes(p |> type(Point))
  p$x <- p$x + 1
  return(p)
}
fcpp <- ast2ast::translate(f, types_f = types_f)
p <- structure(list(x = 1, y = 2), class = "Point")
fcpp(p)

# Inner functions and uniroot
# ------------------------------------------------------------------
f <- function(interval) {
  argtypes(interval |> type(vec(double)))
  g <- fn(
    argtypes(x |> type(double)),
    return(double),
    {
      return(x^2 - 4)
    }
  )
  res <- uniroot(g, interval, 1e-10, 1000)
  return(res$root)
}
fcpp <- ast2ast::translate(f)
fcpp(c(0, 10))

# External pointer interface
# ------------------------------------------------------------------
f <- function() {
  print("Hello World from C++")
```

```
}
ptr <- ast2ast::translate(f, output = "XPtr")

# Call from C++ side
Rcpp::sourceCpp(code = "
  #include <Rcpp.h>
  typedef void (*fp)();
  // [[Rcpp::export]]
  void call_fct(Rcpp::XPtr<fp> inp) {
    fp f = *inp;
    f();
  }")
call_fct(ptr)


## End(Not run)
```

Index

make_package, 2

translate, *2*, *3*, 4