nCompiler User Manual

Table of contents

Introduction

Part I

nFunctions

1 Introduction to nFunctions

1.1 Quick summary

- An `nFunction` lets you compile a subset of R code, mostly numerical, via C++ without knowing C++.

1.2 Introduction to `nFunction`

An `nFunction` is a function in R that can be compiled via C++. You do not need to know C++ to create and use an `nFunction`. If you do know some C++, including Rcpp, you can include your own chunks of C++.

1.2.1 Quick example

Here is a simple example. We will:

- write an R function to multiply its input by 2 and return the result.
- create an `nFunction` from the R function with declarations of input and return types.
- use the function *uncompiled* (natively in R) or *compiled* (via code-generated and compiled C++).

```
mult2 <- function(x) {
  ans <- 2 * x
  return(ans)
}

nf_mult2 <- nFunction(
  fun = mult2,
  argTypes = list(x = 'numericVector'),
  returnType = 'numericVector'
)
nf_mult2(1:3)
```

```
[1] 2 4 6
```

```
Cnf_mult2 <- nCompile(nf_mult2)
Cnf_mult2(1:3)
```

```
[1] 2 4 6
```

1.2.2 General features

An `nFunction` has the following features:

- Arguments and returned objects must have declared types.
- Arguments can be passed by copy (default), reference, or block reference.
- A variable can only have one type. It can't be re-used as a different type as it can in R.
- Code is limited to a subset of R, primarily math and basic flow control (e.g. for loops and if-then-else). This subset of R can be automatically converted to C++.
- An `nFunction` can be run uncompiled (as R code, usually for debugging) or compiled. Uncompiled and compiled behaviors are identical most of the time, but there are exceptions.
- Code can also include hand-coded chunks of C++ and Rcpp, enabling a lot of flexibility. If there is hand-coded C++, the code can't be run uncompiled. If you want to mix hand-coded C++ with automatically generated C++, you may need to learn a bit about the types involved.

Each of these points is covered in a subsection.

> **Check out this note.**
>
> I included this as an example of a callout.

2 types for inputs and outputs

2.1 Quick summary

- You need to say what the types of inputs and outputs will be.
- Usually the types of variables created within functions will be handled automatically.
- Types include basic types (scalars, vectors, matrices, or arrays of doubles, integers, or logicals), `nClass` types, `nList` types, and `Rcpp` types, among others.

> **!** nCompiler supports true scalars
>
> In R, there are no true scalar. Rather, what feel like scalars are length-1 vectors. In `nCompiler`, generated C++ can include true scalars, so there is an important type distinction between true scalars and non-scalars.

2.2 How to use type declarations

Type declarations are needed to cross the threshold between R and C++. R allows variables to be any type (dynamic typing), while C++ requires each variable to have a declared, unchanging type (static typing).

2.2.1 Two places for type declarations

Type declarations can be given:

- directly in the function code, or
- in separate arguments to `nFunction`.

The `nf_mult2` example given above used the second approach. Here is the alternative:

```
nf_mult2 <- nFunction(
  fun = function(x = 'numericVector') {
    ans <- 2 * x
    return(ans)
```

```
  returnType('numericVector') # This can appear anywhere.
})
```

It is also fine to have text like a function call, e.g. `'numericVector()'` because sometimes one wants to include arguments.

The `returnType` call can appear anywhere in the function body. It does not affect code execution.

2.2.2 Two syntaxes for types

Types declarations can be given as:

- character strings, or
- code (sometimes "quoted").

The examples so far use character strings. Here are examples with code:

```
nf_mult2 <- nFunction(
  fun = function(x = numericVector()) {
    ans <- 2 * x
    return(ans)
    returnType(numericVector())
  })
```

or

```
nf_mult2 <- nFunction(
  fun = mult2,
  argTypes = list(x = quote(numericVector())),
  returnType = quote(numericVector())
)
```

2.2.3 Type objects

Sometimes it is useful to make an object that holds the type, allowing one to write code to construct `nFunctions`. That can be done with the function `nMakeType` and the special syntax `T` in type declarations. For example:

```
my_type <- nMakeType(numericVector())
nf_mult2 <- nFunction(
  fun = function(x = T(my_type)) {
    ans <- 2 * x
    return(ans)
  },
  returnType = quote(T(my_type))
)
```

For illustration, this uses `my_type` once in the function code and again in the `returnType` argument.

> The system for type objects may change.
>
> The scheme for making and using objects containing type information is still being designed and so might change.

2.2.4 Numeric types

A numeric type in `nCompiler` comprises the number of dimensions and the scalar type of each element. A special type for sparse matrices is also provided.

- "Number of dimensions" really means the number of *index* dimensions. For example a matrix or 2D array has two (index) dimensions and a 3D array has three index dimensions. [1]
- The scalar type of each element can be `numeric`, `integer`, or `logical`. A synonym for `numeric` is `double` (for standard "double precision" numbers). Note that in R, "numeric" means "integer or double" (e.g. try `is.numeric`), but in `nCompiler` type declarations, `numeric` means `double`.

2.2.4.1 Declaring numeric types

There are several ways to declare numeric types:

- `integerScalar()`, `numericScalar()`, and `logicalScalar()` are what they sound like.
- `nScalar(type="integer")` is the same as `integerScalar()`. The `type` argument defaults to `"double"` and can also be `"logical"`. (The "n" in "`nScalar`" and similar names below stands for "nCompiler", not "numeric".)

[1] Of course, mathematically a vector of length n represents a point in n-dimensional space. That's not what we mean by number of dimensions. A vector is considered one-dimensional because elements are identified by just one index.

- `numericVector()`, `nVector()`, and `double(1)` are all double-precision vectors.
- `integerVector()`, `nVector(type="integer")` and `integer(1)` are all integer vectors.
- `logicalVector()`, `nVector(type="logical")` and `logical(1)` are all logical vectors.
- The `*Vector()` forms and `nVector` can take a `length` argument. `nVector` can also take a `value` argument. `double(1)` exists for compatibility with `nimble`'s type system. The forms like `double(1)` exist for compatibility with `nimble`'s type system.
- The prefixes "`numeric`", "`integer`", and "`logical`" can also go with the suffixes "`Matrix`" or "`Array`". For example: `integerArray(nDim=3)`.
- `nMatrix` and `nArray` are similar but take the element type (`double`, `integer`, or `logical` are an argument).

2.2.5 Character types

- `string`

2.2.6 Sparse matrix types

- `nSparseMatrix`
- `nSparseVector`

2.2.7 Rcpp types

Many Rcpp types are supported. Using these requires some of your own C++ coding. These include:

- `RcppEnvironment`
- `RcppList`
- `Rcpp<Numeric, Integer, Character, Logical, Complex><Vector, Matrix>` (e.g. `RcppCharacterVector`).
- Others to be listed later.

2.2.8 C++ types

- `SEXP`
- `void`
- `cppVar`
- `nCpp` (for direct coding of an arbitrary type)

2.2.9 nClass and nList types

These will be covered in later sections.

3 types in function code

3.1 Quick summary

- Usually variables inside `nFunction` code will have their types deduced automatically and match R's behavior.
- If you need to take more control, you can do so.

Types of variables created in a function will be automatically determined from the code. For example:

3.2 The type of a variable can't be changed

Once a variable is created, its type can't later be changed.

3.3 The mimic-R rule

To the extent possible, nCompiler mimics R's output types based on input types. There are cases where this is impossible.

3.4 Surprises and manual control over types

Sometimes you may be surprised by the deduced type of a variable.

- declare()
- checking types
- create a variable with an explicit type
- Dimensions
- `nCpp`

3.4.1 Automatic conversions among types when possible

Tip: Sometimes types will differ from R

3.5 To-do:

- Add type inspection features for debugging.

4 Passing arguments by copy, reference or block reference

4.1 Quick summary

- By default, arguments are passed by copy (like R), so that changes to values are only local.
- You can pass arguments by reference (very unlike R), so that changes to values are also seen from the calling function.
- Passing by "block reference" allows changes to values by reference but doesn't allow changing the size. This is useful if, for example, you want to pass `x[11:20, 11:20]` as a matrix by reference.

5 What operations can be compiled?

5.1 Quick summary

- Most math and basic flow control.

5.2 Basic math

5.2.1 Binary functions

- `+`, `-`, `*`, `/`, `%%`

5.2.2 Unary functions:

- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `asinh`, `acosh`, `atanh`, `logit`, `ilogit`, `expit`, `probit`, `iprobit`, `phi`, `cloglog`,`icloglog`, `ceiling`, `floor`, `round`, `trunc`, `lgamma`, `loggam`, `log1p`, `lfactorial`, `logfact`, `mean`, `prod`, `sum`, `exp`, `log`,`sqrt`, `abs`, `cube`, `square`.

5.3 Reduction operators

- `min`, `max`, `all`, `any`, `length`

5.4 Linear algebra

- `%*%`

5.5 Boolean

- `pmin`, `pmax`, `==`, `!=`, `<=`, `>=`, `<`, `>`, `&`, `|`, `+`, `/`, `*`

5.6 Distributions

-`dbeta`, `dbinom`, `ddexp`, `dgamma`, `dinvgamma`, `dlnorm`, `dnbinom`, `dnorm`, `dt`, `dt_nonstandard`, `dunif`, `dweibull`

Tip: note on recycling rule

5.7 Flow control:

- if-then-else
- integer for loops

5.8 To-do:

- more general for loops
- add math operators
- add distributions

6 Controlling compilation

There are many fine-grained controls over compilation that can be provided in the `compileInfo` list. These include:

-

Part II

nClasses

Part III

Developer manual

7 Basic objects for nFunctions and nClasses

7.1 nFunctions

- An `nFunction` (e.g. "`nf`") is an object of class `nFunctionClass`, which inherits from ("contains") R's base `function` class. This means an `nFunction` **is a function** and also has slots for `internals` and `originalCode`.
- The `internals` of `nFunction` is an object of class `NF_InternalsClass`. Access should be via `NFinternals(nf)` and `NFinternals(nf) <-`.
- When it is time to compile, an `NF_CompilerClass` object is created. This has the purpose of creating a *cppDef*, which for an `nFunction` will be a `cpp_nFunctionClass` object.
- The `compileInfo` argument to `nFunction` provides an arbitrary list of compilation-relevant information that will be passed through the various steps of compilation, sometimes with additions or modifications along the way.

7.2 nClasses

- An `nClass` (e.g. "nc") is an `R6` class. The class object is called the *generator* because it can create objects of the class (e.g. `nc$new()`).
- An `nClass` has internal information stored in an `NC_InternalsClass` object that can be accessed by `NCinternals(nc)` and `NCinternals(nc) <-`.
- `Cpublic` methods of the `nClass` are `nFunctions`. (As for `R6` classes, if they are defined before being included in the class definition, their closure is changed when added to the class. All methods of a class object will share the same closure, set by the `env` argument to `nClass`.)
- When it is time to compile, an `NC_CompilerClass` object is created. This has the purpose of creating a *cppDef*, which for an `nClass` will be a `cpp_nClassClass` object.
- As for `nFunctions`, the `compileInfo` argument to `nClass` provides an arbitrary list of compilation-relevant information that will be passed through the various steps of compilation, sometimes with additions or modifications along the way

For both `nFunctions` and `nClasses`, the resulting *cppDef* objects (of which there are other types as well) have the final purpose of creating content for `Rcpp_packet` objects. These contain the actual text pieces to be written to .cpp and .h files.

8 Developer: nCompile overview

`nCompile` is the entry point for all compilation.

8.1 Direct, package, and package development modes

There are three ways `nCompile` can organize generated source code:

- `nCompile` with `package=FALSE` results in separately ("directly") generated code with no package structure. This is the fastest way to compile, so it is good for development iterations. One can't really serialize and save objects in a useful way in this mode, because there is no package to load in order to support loading the saved objects later.
- `nCompile` with `package=TRUE` results in code generated into a package structure by calling `writePackage` internally. This is slower but does allow serializing and saving objects and later re-using them. It can be used as an "under-the-hood" way to use package infrastructure to support useful behavior even when a user is not aiming to develop a package.
- `writePackage` supports generating code into a package that a user is developing. One can then compile the package using normal mechanisms.

8.2 Following nCompile code is tricky

- `nCompile` attempts to handle inputs fairly generally (e.g. are they named versus using names when they were defined by `nFunction` or `nClass`; are interface settings to be over-ridden; and more).
- In "direct" mode, the code is fairly linear.
- In "package" mode, the code is tricky. `nCompile` calls `writePackage`, which calls `nCompile` with different inputs that go down different conditionals, which then return to `writePackage`, which then returns to the original `nCompile` call.

8.3 Current issues:

- Currently, if one unit needs another (e.g. one `nFunction` calls another `nFunction`), both **must** be provided in the same call to `nCompile`. For some purposes (e.g. support for nimble), we may need to have a mode where needed units are automatically found and included.
- The ordering of `#define` and `#include` statements has been particularly tricky, partly because of some ways that Rcpp works (due to features of C++). This has been iteratively generalized and may need further generalization. In general, this work involves `Rcpp_packet` objects and how simple vs. complicated they need to be.

9 Developer: Compilation stages

Processing code of an `nFunction` goes through a series of stages (see `processNFstages`). Some of these were drafted into the system but either not used or not used as originally intended. The compiler stages have numeric codes and are listed in `NFcompilerStages`.

- "setInputOutputTypes": The actually just creates the initial `symbolTable` based on function arguments.
- "substituteMangledArgumentNames": This replaces an argument name like `log` with a name like `ARG1_log`. This step is done only for argument names that clash with keywords. In R, a function name can also be used as a variable name, but that is not the case in C++.
  - Lack of always checking if an argument name needs mangling or demangling may be a source of future rare bugs.
- "initializeCode": This uses `nParse` to parse code into nCompiler's own abstract syntax tree objects, called `exprClass` objects.
- "initializeAuxiliaryEnvironment": The `auxEnv` object is an environment used for sharing information across "handlers" in later stages. For example, it collects information about what other `nFunction` or `nClass` definitions are needed by the current one, what derivative or parallelization information is needed, whether an `nList` is used, what the names of function arguments are, and other such information.
- "normalizeCalls": This is similar to `match.call` and is used to put arguments in order and fill in defaults. It uses the `matchDef` element of the operator definition ("opDef"), if available. It moves compile-time arguments out of the argument list and into the `aux` field of the `exprClass` object. It also has the first of a set of environments for *handlers* of keywords at different compilation stages. Its environment is called `normalizeCallsEnv`. If the opDef's entry for `normalizeCalls` contains a `handler` entry, the function named by that entry will be found in the `normalizeCallsEnv`. There aren't many keywords needing handlers, but an important one is that calls to an `nFunction` or an `nClass` method are replaced with 'NF_CALL_(<method_name>, <arguments…>)'.
- "simpleTransformations": This makes transformations of code that do not require information or processing about argument types. Handlers can be provided as the `opDef$simpleTransformations$handler`, which will be found in `simpleTransformationsEnv`. Examples include simply replacing one keyword name

with another for later processing as well as processing `cppLiteral` or its more general version, `nCpp`. These are tools for including hand-written arbitrary C++ anywhere one wants.

- "buildIntermediateCalls": This "lifts" a call from a longer expression to create an intermediate variable first. This is done here only for calls that always needs such lifting, and there aren't many of those. Currently the list includes `eigen`, `chol`, and `run.time`. It is possible that eventually this stage will not be needed.
- "labelAbstractTypes": This is one of the beating hearts of the whole compilation system. It labels every step of the AST with type information. In the predecessor nimble compiler, the analogous step ended up having many additional purposes cobbled onto it because it usually has its hands on key information about a piece of syntax. In `nCompiler`, the goal is to maintain better discipline about what belongs in this step and what doesn't. In particular we should try to avoid implementation-specific steps, e.g. steps specific to the Eigen (or other) library; this is for *abstract* types, not implementations. The `opDef$labelAbstractTypes$handler` can name a handler that will be found in `labelAbstractTypesEnv`. The handler should fill in the `type` field of each `exprClass` object. Sometimes additional information is collected or managed when it is directly related to types.
- "processAD": This does processing related to automatic differentiation. Note that AD types and functions are first-class objects at this point. More will be described in another section.
- "addInsertions": This inserts new lines of code created and collected by previous steps. This has not been necessary to use much.
- "setImplementation": The idea for this was that we currently use Eigen for non-scalar and linear algebra, but conceivable in the future we could use other implementations. In practice currently this step doesn't do much meaningful.
- "doImplementation": This is where "eigenization" happens.
- "finalTransformations": I don't think much is done here.
- "addDebugging": This is used only if C++ debugging is turned on.

10 Developer: types and symbol tables

10.1 Type symbols

The type of a symbol (variable) is represented by an R6 object. Normally these inherit from `symbolBase`. The fields of `symbolBase`, representing common needs of many types, are:

- `name`: This is only needed for an actual variable, not for example for a return type.
- `type`: A string label for the type.
- `isRef`: logical for whether the type is handled by reference.
- `isArg`: logical for whether the type is a function (or method) argument.
- `interface`: logical for whether the type can be interfaced between R and C++. (This may turn out to not be needed much.)
- `implementation`: arbitrary information about C++ implementation. (This may not be needed at all.)

In addition, the base methods are:

- `initialize`
- `shortPrint`: This is used by `exprClass$print` (`print` is standard method name for an `R6` class, similar to `show` for base R classes).
- `generateUse`: This is used for C++ generation.

10.2 Generating C++

Normally a symbol represents an abstract type and the symbol object has a method `generateCppVar` that returns an object inheriting from `cppVarClass`, which represents the C++ type used to implement the abstract type.

10.3 Symbol tables

Symbol tables are represented by an object of type `symbolTableClass`. This has methods for adding, accessing, and removing symbol objects. It also have a parent symbol table (`parentST`), which represents its scope. For example, in an `nFunction`, there will be a symbol table for the function arguments. This will be the parent of another symbol table for local variables. If the `nFunction` is a method of an `nClass`, the parent of the argument symbol table will be yet another symbol table for the class member variables and methods.

10.4 From type declarations to type symbols

A type declaration takes a form like the character string `"nVector(type='integer')"` or the code `nVector(type='integer')`. The function `argType2symbol` converts a type declaration (and possibly other information) into a type symbol. Since multiples types to compose a symbol table often start as a list of type declarations, there is a function `argTypeList2symbolTable`.

`argType2symbol` takes the following steps:

- If the type declaration is already a symbol object, return a named clone of it.
- If the type declarations is character, parse it.
- Look for a handler in the `typeDeclarationList` object. This is the central source for creating types from declarations.
- If found, call the handler, which will return a symbol object. The handler list contains many type synonyms (e.g. `numericVector`, `nVector(type="numeric")`, etc.) that may return the same underlying symbol object.
- Try to find an object and deduce the type from the object.
- If nothing has worked so far, set it as a "to-be-determined" type, represented by a symbol object of class `symbolTBD`. For example, this case covers `nClass` types whose definitions are not currently available.

10.5 Some of the symbol types available

- `symbolBasic`: For all basic numeric types (integer, double or logical with some number of index dimensions).
- `symbolBlank`:
- `symbolNF`: For an `nFunction`.

- `symbolTBD:` to-be-determined during a compilation stage when other type can be found.
- `symbolNC`: For an `nClass` object.
- `symbolNCgenerator` : For an `nClass` generator
- `symbolNlist` : For an `nList` object
- `symbolRcppType` : For an Rcpp type, for which there are some derived types for specific Rcpp types.
- `symbolSparse` : For sparse matrices.
- `symbolCppVar`: For a C++ type. (In C++ code generation, types are represented by `cppVarClass` objects. The `symbolCppVar` is different: it is for an abstract type that declared to be a C++ type and hence can't be processed through compilation except for being used in hand-coded C++.)

11 Developer: the expression class

In R, code itself is an object, and can be accessed like a list, but there isn't a good way to hold additional information. Therefore `nCompiler` uses its own class, `exprClass`, to represent code expressions. An `exprClass` object has fields available for lots of useful information.

Consider `foo(a = 1, b = x, c = bar(y))`. We obtain the `exprClass` representation of this by:

```
AST <- nParse(quote(foo(a=1, b=x, c=bar(y))))
```

The input could alternatively be the string "foo(a=1, b=x, c=bar(y))"

Here is a guide to the `exprClass` object `AST` (for abstract syntax tree):

```
AST      # a print method attempts to show the syntax tree
```

```
foo
  a=1
  b=x
  c=bar
    y
```

```
AST$name # name is the name of the function
```

```
[1] "foo"
```

```
AST$args # args is a list of other exprClass objects for the arguments
```

```
$a
1

$b
x

$c
bar
  y
```

```
AST$isCall              # TRUE since the expression is a function call.
```

```
[1] TRUE
```

```
AST$Rexpr               # The original R expression
```

```
foo(a = 1, b = x, c = bar(y))
```

```
AST$args[[1]]$isLiteral # TRUE since 1 is a "literal" value.
```

```
[1] TRUE
```

```
AST$args[[1]]$name      # name for a literal gives its value
```

```
[1] 1
```

```
AST$args[[2]]$isName    # TRUE since `x` is a name but not a call
```

```
[1] TRUE
```

```
AST$args[[2]]$name      # name for a name gives the name as a string
```

```
[1] "x"
```

```
  AST$args[[2]]           # From the `print` method, we see the string unquoted.
```

```
x
```

```
  AST$args[[3]]$isCall    # TRUE
```

```
[1] TRUE
```

```
  AST$args[[3]]$args[[1]] # exprClasses can become nested to a high degree.
```

```
y
```

Note that in R, `{` is itself simply a function. Each line of code before the closing `}` is an argument, it accepts arbitrarily many arguments, and it returns the value of the last argument. Hence, an entire code block can be nested in an `exprClass`.

```
  ASTblock <- nParse(quote(
    {
      a <- 1
      b <- a + 2
      print("hello world")
    }
  ))
  ASTblock$name
```

```
[1] "{"
```

```
  ASTblock$args[[1]]
```

```
<-
  a
  1
```

```
  # etc.
```

Assignments are special operations, so they get marked specifically:

```
ASTblock$args[[1]]$isAssign # TRUE because the operator is `<-`, `=`, or `<<-`.
```

```
[1] TRUE
```

`exprClass` objects are doubly-linked, meaning an object for a call knows about its arguments and the arguments correspondingly know about the call they are part of. Conceptually, there are arrows both up and down the syntax tree. For processing, one can recurse down a tree or up a tree.

```
AST
```

```
foo
  a=1
  b=x
  c=bar
    y
```

```
arg_c <- AST$args[['c']]
arg_c
```

```
bar
  y
```

```
arg_c$caller      # This is AST
```

```
foo
  a=1
  b=x
  c=bar
    y
```

```
arg_c$callerArgID # 3 because this is the third argument of the caller
```

```
[1] 3
```

This means that one can't arbitrarily or naively set arguments, because they will not be doubly linked. When an incorrectly formed `exprClass` object is printed, it will indicate that something is wrong.

```
ASTcopy <- nParse(quote(foo(a=1, b=x, c=bar(y))))
ASTcopy$args[[2]] <- nParse(quote(z)) # DANGER, DON'T DO THIS.
ASTcopy # We get a warning that the second argument is not doubly linked correctly.
```

```
foo
  a=1
  b=z
****Warning: caller and/or callerArgID are not set correctly.
  c=bar
    y
```

There is a set of functions for manipulating `exprClass` objects. (They are not exported.) For example:

```
ASTcopy <- nCompiler:::copyExprClass(AST)
nCompiler:::setArg(ASTcopy, 2, nParse(quote(z)))
ASTcopy
```

```
foo
  a=1
  b=z
  c=bar
    y
```

Some of the functions for manipulating `exprClass` objects include:

- `copyExprClass(original)` : Make a deep copy that sets up the double links.
- `insertExprClassLayer(expr, 2, 'g')` Make the second argument of `expr` be wrapped in `g()`.
- `removeExprClassLayer(expr, argID)`. For the caller of `expr`, replace the argument that is `expr` with the `argID` argument of `expr`. Example: If `expr` is `inner(a, b)` and its caller is `outer(inner(a, b))`, then `removeExprClassLayer(expr, 2)` will replace `inner(a, b)` with `b`, so that the caller is now `outer(b)`.
- `wrapExprClassOperator(expr, 'g')` Wrap the entire AST `expr` inside `g()`.

- `newBracketExpr(args)`. Wrap a list of `exprClass` objects (args) inside `{}`.
- `setCaller(value, expr, ID)` Make `value` be the `ID` argument of `expr`.
- `insertArg(expr, ID, value)` Insert `value` as a **new** argument in position ID for `expr`, shifting other arguments up one position.
- `setArg(expr, ID, value)` Set `value` as the argument in position `ID` for `expr` (replacing whatever is previously in that position). This uses `setCaller` but is more general because `ID` can be a string and some error-trapping is done.
- `removeArg(expr, ID)` Remove the argument in position `ID` from `expr`.

Many of the above functions can take additional arguments that are not shown. Some of them support `ID` to be either a position number or a name of an argument.

11.0.1 Fields used during compilation

A key step in compilation is labeling every call in the abstract syntax tree of an `nFunction` body with its type information. This is done during the "labelAbstractTypes" stage of `nFunction` compilation. Here is some information on that and other fields used during compilation:

- `type` This field will be populated with a symbol object representing the type of the `exprClass`. For a literal, this is the type of the literal. For a call, this is the type returned by the call.
- `aux` This field is used for any auxiliary information that any compilation stage needs to attach to an `exprClass` object for later use.
- `insertions` This field holds other `exprClass` objects that will need to be inserted into the full code body in order to implement its expression.
- `cppADcode`: This is `TRUE` if the `exprClass` represents an expression used in automatic differentiation.

There are also some functions (and room for more) to support more concise coding of common `exprClass` objects that need to be created during compilation, with the `type` field already populated. For example, sometimes we need to insert a literal and do so after type labeling. Since it is expected that all `exprClass` objects have types labeled at that and subsequent points, any inserted `exprClass` objects at later points must be inserted with the `type` field populated. We have `literalDoubleExpr`, `literalIntegerExpr`, and `literalLogicalExpr` to create type-labeled literal `exprClass` objects, which otherwise takes a few lines of verbose code.

12 Developer: Operator definitions

An operator definition, or "opDef", provides information about how each stage of compilation should be handled for each operator (i.e. keyword) that can be compiled.

The opDefs can all be found in the `operatorDefEnv` environment, which provides a single place to determine how an operator is processed.

> ❗ Be careful when operator names are changed during processing.
>
> Sometimes, handling an operator will result in changing its name. When that happens, the new name will be used to look up the relevant handling for later compilation stages.

An opDef contains some basic information and then fields for each compilation stage. For example, here is the opDef used for both `+` and `-` :

```
list(
  labelAbstractTypes = list(
    handler = 'BinaryUnaryCwise',
    returnTypeCode = returnTypeCodes$promoteNoLogical),
  eigenImpl = list(
    handler = 'cWiseAddSub',
    replacements = list(
      '+' = list(
        'LHS' = 'nCompiler::binaryOpReshapeLHS<nCompiler::plus>',
        'RHS' = 'nCompiler::binaryOpReshapeRHS<nCompiler::plus>'
      ),
      '-' = list(
        'LHS' = 'nCompiler::binaryOpReshapeLHS<nCompiler::minus>',
        'RHS' = 'nCompiler::binaryOpReshapeRHS<nCompiler::minus>'
      )
    )
  ),
  cppOutput = list(
    handler = 'BinaryOrUnary')
)
```

In each compilation stage, if there is a `handler` entry, that will be called, and the entire list for that stage may be passed as an argument to the handler. For example, in the "labelAbstractTypes" stage, if there is an opDef field `labelAbstractTypes`, and if it has a `handler` entry, the named function is called. In this case, "BinaryUnaryCwise" is called (named as such because `+` and `-` may be unary or binary operators and are component-wise operators). The entire `labelAbstractTypes` list will be passed as the `handlingInfo` argument to the handler.

12.1 Check the source code for handler argument protocols.

Each compilation stage may use a different standard set of arguments that will be passed to handlers in that stage.

A common need for "labelAbstractTypes" is to determine the type of a return object from a numerical operation. The `returnTypeCode` provides an option for that. See below for more information.

12.2 The `help` field

A help field in an opDef can provide a simple help string. This is not uniformly filled in and is not currently utilized, but it may be in the future.

12.3 The `matchDef` field for argument normalization

`matchDef` is a special field that is used during both the "normalizeCalls" stage and parsing by `nParse`. It is included as its own field in an opDef rather than nested within the "normalizeCalls" field. The `matchDef` provides an R function prototype (with empty body, `{}`), which is used for its argument order, names, and initial values.

`matchDef` is similar to `match.call` in R, but has some different needs. A common step in processing a call in R, if one wants to use the code provided as arguments rather than just their values, is `match.call`. For example:

```
match.call(function(a = 1, b, c = 5) {}, quote(foo(c = 100, 20)))
```

```
foo(a = 20, c = 100)
```

Here the first argument is used as a function prototype, and the second argument is a call that needs to be normalized to the prototype: arguments are ordered. Often this is used inside a function in R, and the arguments default to the actual function prototype and the actual call of the function, respectively, and thus are both omitted. Notice that since `b` has no default in the prototype, it is omitted from the result.

`nCompiler` needs a distinct version of this kind of feature for several reasons. It must work with `exprClass` objects. All arguments must be filled in, even if no value is provided, because at some point in the path towards C++ code one can't simply have missing arguments (although processing steps must still decide what to fill in if any values are missing). Auxiliary information is recorded about whether arguments were missing from the call. And compile-time arguments are separated. The last two points need explanation.

12.3.0.1 Missing values

Say a line of code is `foo(a = 10)` for a prototype `function(a = 1, b = 2)`, resulting in the normalized call `foo(a = 10, b = 2)`. It may be helpful in later processing to know whether each argument was provided or was filled in by a default value. In addition, it is useful to know if any arguments are completely missing, neither provided nor with a default. This information is stored in two entries in the `aux` list of the `exprClass` for `foo`, illustrated below.

12.3.0.2 Compile-time arguments

The opDef entry `compileArgs` can give a character vector of any argument names that can only be processed at compile time and then can never change. An example is `x <- nVector(type = "integer")`. The `type` argument can't be a variable that changes from run to run (i.e. at run-time). It is a compile-time argument. In a case like this, we would see `compileArgs="type"` in the opDef. The `compileArgs` are separated at the earliest possible step, namely `nParse`, and are placed in a `compileArgs` field in the the `aux` list of the resulting `exprClass`.

12.3.0.3 Example

Here is an example of the whole `matchDef` and `compileArgs` system.

Say we have an opDef for an operator `foo`:

```
opDefEnv <- list2env(
  list(foo = list(
    matchDef = function(a=1, b, c=2, d=x){},
    compileArgs = 'd'
    ))
```

```
)
```

and say we have the expression

```
code <- quote(foo(c=100, d=x2))
```

We can manually imitate how these will be handled. The AST from code will be:

```
expr <- nParse(code, opDefEnv = opDefEnv)
expr
```

```
foo
  c=100
  NULL
```

```
expr$args
```

```
$c
100

$d
NULL
```

```
expr$aux$compileArgs
```

```
$d
x2
```

So far, the compile-time argument `d` has been moved out of the call and into the auxiliary information. Its place is held by a `NULL`.

Note that the value of `d` can be a variable, and a compiler stage can use scoping to find the value of `x2`, but that value can only be resolved at compile-time, once.

Later, when the "normalizeCalls" stage is done, the use of `matchDef` is done like this:

```
matched_expr <- nCompiler:::exprClass_put_args_in_order(opDefEnv$foo$matchDef,
                                                        expr)
```

```
  matched_expr                          # defaults filled in; arguments ordered.
```

```
foo
  a=1
  c=100
  NULL
```

```
  matched_expr$aux$missing              # b is entirely missing, without default
```

```
[1] "b"
```

```
  matched_expr$aux$provided_as_missing # a and b were not provided in the call.
```

```
[1] "a" "b"
```

Note that some (at the time of this writing, *many*) handlers were written before this full set of information was available in the `exprClass` objects. As a result, they may inspect these objects in redundant or inconsistent ways, which we intend to clean up over time.

12.4 More details on each compilation stage

12.4.1 `normalizeCalls`

When an `nFunction` call is found during `normalizeCalls`, the opDef for the name "`NFCALL_`" is used. When an `nClass` method call accessed as a local call (i.e. from an object of the `nClass`) is found, the opDef for "`NCMETHOD_`" is used. In both cases, the handler is `nFunction_or_method_call`.

For an `nFunction` (or method) called "`foo`", this results in changing

- `foo(a, b)` to
- `NFCALL_(FUN_ = foo_cpp_name, a, b)`,

where "`foo_cpp_name`" is the C++ function name for `foo`. As a result, subsequent handlers will be found from the "`NFCALL_`" opDef. In addition, the `exprClass` object for the argument `foo_cpp_name` has several fields added to its `aux` list:

- `obj_internals` will be the `NF_InteralsClass` for `foo`, i.e. the result of `NFinternals(foo)`. This allows one to look up nearly everything about `foo`.
- `nFunctionName` will be the R name, i.e. "foo".

Note that accessing methods from other `nClass` objects, such as by `myObject$foo(a, b)` is handled during `labelAbstractTypes`, in the handler for "`$`", which can inspect the type of `myObject`.

12.4.2 `simpleTransformations`

This stage has only a few handlers:

- `replace` uses the `handlingInfo` field `replacementName` and replaces the operator name with the value of `replacementName`.
- `minMax` changes `min` or `max` to `pairmin` or `pairmax` if there are two arguments.
- `Literal` is used for `cppLiteral` and `nCpp` to evaluate the `text` argument in the appropriate scope.

12.4.3 `labelAbstractTypes` and the `returnTypeCodes` system

- This has too many handlers to list here.
- The `returnTypeCodes` system for determining numeric outputs from numeric inputs works as follows:
  - `AD`, `double`, `integer`, or `logical` name the type always returned by an operator. Their information content is ranked in that order.
  - `promote` means the type of the highest-information argument will be returned.
  - `promoteToDoubleOrAD` means the return type will be `double` unless there is an `AD` argument, in which case it will be `AD`.
  - `promoteNoLogical` means the type of the highest-information argument will be returned, with the exception that `logical` is promoted to `integer`.

There are some common transformations that are done during `labelAbstractTypes` (which can't be done during `simpleTransformations` because they rely on type information):

12.4.4 `eigenImpl`

- This has too many handlers to cover here.

12.5 `cppOutput`

- This has too many handlers to cover here.

12.6

13 Providing operator definitions for new keywords or nClass methods

`nCompiler` is designed to be extensible.

13.1 Defining how new keywords should be handled

If you want to support a new keyword for compilation, you can do so by providing an operator definition, including handlers for any compilation stage(s) where they are needed. This is done by `registerOpDef` and can be removed by `deregisterOpDef`.

A difference from built-in handlers is that a new handler can be provided as a function rather than just the name of a function.

13.2 Defining how `nClass` methods should be handled.

This is an idea not yet implemented.

It would be nice to allow controllable and extensible handling of any method (`nFunction`) within an `nClass`.

14 Developer: C++ definition classes

15 Developer: C++ implementations