Specifications for a Julia package providing ROOT TFile write support

Ph. Gras[1] and Pere Mato[2]

[1] *IRFU, CEA, Université Paris-Saclay, Gif-sur-Yevette, France*
[2] *CERN, European Organization for Nuclear Research, Geneva, Switzerland*

Revisions

- updated `Write()` function interface on July 12, 2024

1 Introduction

The ROOT framework provides a file format suited to store the large data sets produced in HEP (hundreds of Petabytes of data), in particular at the CERN Large Hadron Collider. It provides a fast access, a structured data organization with columnar data support, compression, and self-contained data model description. Support for this format is essential for HEP application.

The UnROOT package, implemented purely in Julia, provides support allows reading the file in ROOT format. Write support is lacking from the Julia ecosystem. Its implementation is more difficult than for the read support. For this reason, the development plan of the UnROOT package is to limit the package for reading files.

The Julia wrapper, ROOT.jl for ROOT coming with the WrapIt! tool as a usage example provides ROOT file write access. We will continue to call this wrapper ROOT.jl in the rest of this document and it should not be confused with the homonym package from JuliaHEP/ROOT.jl that works with old version of Julia only. The ROOT I/O system is a versatile serialization tools that support writing an arbitrary number of instances of arbitrary types, from simple type to complex classes that can count pointers and references among their fields. The objects can be organized in the file in a tree structure using folders similarly to a file system.

ROOT.jl supports write for all the ROOT classes it wraps. To write instances of other classes e.g., a user defined one, it is necessary to wrap the classes, since it needs to be instantiated from the Julia code.

The ROOT `TTree` class plays a special role. It is used to store N instances, called *entries*, of the same data structure. For the usage in the High energy physics (HEP) research, where statistical analyses of many observations e.g., of particle beam collisions, are performed, one instance corresponds to one observation, called *physics event* and is also an event in the statistical terminology.

`TTree` uses a columnar storage. It can be viewed as a Table –in the Tables.jl sense– or a Dataset –in the R language, Pandas and DataFrames.jl sense, where a row corresponds to a `TTree` entry or physics event.

Nevertheless, a TTree is often more complex that a Table used in other fields. Indeed, data stored in a row can have a tree structure, hence the name `TTree`, the first `T` is letter the ROOT framework uses to prefix all the C++ classes it provides. In the analogy with tables, we can see the first level of branches in the tree hierarchy as the table columns. When a top branch has sub-branches, then the data stored in the corresponding column has itself a tree structure.

The simplest ROOT-based data formats used by the ATLAS and CMS experiments at the CERN Large hadron collider (LHC), called nanoAODs (each experiment has its own format), each top-level branch (or table/dataframe column) contains either a scalar or a homogeneous vector (that is a one-dimension array of scalar of the same type). On the other hands, the CMS RECO and AOD formats is an example of a sophisticated ROOT-based data format.

While ROOT.jl provides Julia bindings to the C++ ROOT I/O interface and in particular to build and write `TTree`s, this interface lack user-friendliness and "crash" safety typically expected from a Julia package.

This document describes the higher-level Julia interface we would like to provide to the user thanks to a Julia package that will use ROOT.jl.

We will consider a baseline specification, that will cover the minimal data structure complexity the developed package must support and some extensions, called *feature package*, that can be implemented depending on the project advancement.

2 Baseline write interface specification

`TFile` instantiation

Creation of a `TFile` will be done using the `ROOT.TFile` function provided by ROOT.JL, that is the plain binding of the C++ `TFile` constructor.

`TTree` instantiation

The high level interface will wrap the TTree CxxWrap wrapper in a struct that will hold the type of data stored in each TTree entry (≡ row). [1]

The `TTree` will be instantiated by a constructor (implemented in Julia), `TTree(dir, name, title, type)`. In case the value passed as third argument is not a data type, a `TTree` with row type identical to the type of the argument will be built [2].

The extra `dir` parameter will be used to set the directory (or `TFile`), where the data must be written. This differs from the C++ API, which uses the current ROOT Directory e.g., set when opening a `TFile`. Explicitly specifying the file without depending on a global state (the current directory) seems a more common approach in computing and hence adopted here.

Filling one `TTree` entry: row-by-row writing

The C++ `TTree::Fill()` function will be replaced by `Fill(tree, row)` in the Julia high-level interface. This function willl add a row to the TTree or, in other words, write a row on disk.

Filling one `TTree` entry: multirow writing

In case the type passed as `rowtype` argument of the `Tree` constructor is a `Table` (which include `DataFrame` from DataFrames.jl), then the `Fill` functions will write each table row as a `TTree` entry.

[1] We could actually keep track of this type as a parameter of the type. Both options will be eventually studied: use of a parameter type may allow run time speed-up, but it can introduce lags from JIT compilation (functions taking the TTree as argument will be invalidated at each new data type change).

[2] This can be easily be implemented by annotating the third argument for the main implementation, `function TTree(dir, name, title, rowtype::Type); ...; end` and adding the method definition, `TTree(dir, name, title, rowexample) = TTree(dir, name, title, typeof(rowexample))`.

Method `Write(dir, name, title, table)`

The `Write(dir, name, title, table)` method will be a shortcut to instantiate a `TTree` from a `Table` and write it to disk in a single function call.

Supported row types

The interface will support as data type, `struct` with fields of the following type. Every field of the `struct` must be annotated with it type (e.g., `i::Int`).

The field can be of following simple types. Missing types will be added to the C++/Julia ROOT.JL wrapper.

Julia type	C++ type[a]	TTree type code	Description
`String`	`char*`	`C`	a character string.
`Int8`	`Char_t`	`B`	an 8 bit signed integer.
`UInt8`	`UChar_t`	`b`	an 8 bit unsigned integer
`Int16`	`Short_t`	`S`	a 16 bit signed integer
`UInt16`	`UShort_t`	`s`	a 16 bit unsigned integer
`Int32`	`Int_t`	`I`	a 32 bit signed integer
`UInt32`	`UInt_t`	`i`	a 32 bit unsigned integer
`Float32`	`Float_t`	`F`	a 32 bit floating point
`Half32`[b]	`Float16_t`	`f`	32 bits in memory, 16 bits on disk
`Float64`	`Double_t`	`D`	a 64 bit floating point
`Double32`[c]	`Double32_t`	d	64 bits in memory, 32 on disk
`Int64`	`Long64_t`	`L`	a 64 bit signed integer
`UInt64`	`ULong64_t`	`l`	a 64 bit unsigned integer
`Int64`	`Long_t`	`G`	a long signed integer, stored as 64 bit
`UInt64`	`ULong_t`	`g`	a long unsigned integer, stored as 64 bit
`Bool`	`bool`	`O`[d]	a boolean
`StdVector{T}`	`std::vector{T}`	N/A	Vector of elements of any of the above type.

[a] See RtypesCore.h for the `typedefs`

[b] Custom type defined as `struct Half32; Float16; end`. Conversion method to usual number types will be provided.

[c] Custom type defined as `struct Double32; Float64; end`. Conversion method to usula number types will be provided

[d] The letter o, not zero.

We have included all types supported by `TTree`. In a first stage types supported by ROOT.JL at the time of implementation will be supported. Then, ROOT.JL C++ code will be updated to support the missing types.

In addition, support for C-arrays, both with fixed size and with dynamic size will be added. We still need to define the interface. One option could be:

- **Vector of fixed size** Use StaticsArrays;
- **Vector of dynamic size** Define a Julia type that wraps `Vector` and takes a symbol as parameter to designate the struct field holding the vector size. `struct DynVector{T,N} Vector{T}; end` [3]

Note of `Write` function implementation

The `Write` function will need to call `SetAddress` to update the data memory addresses. The cost of calls for large data structure will be evaluated.

Only the `struct` base address and the addresses of vector-type field needs to be set again. Optimization of the number of `SetAddress` calls can benefit from a TTee row type defined as a parameter.

For `Tables`, the same type as for the fields of structs is supported.

3 Feature package 1

Reading `TTree` is supported with an interface consistent with the write interface. Reading back `TTree` produced within the baseline support with UnROOT will also be tested and possible shortcoming of UnROOT will be reported to the package developers.

Priority of this extension with respect to the next one remains to be defined.

4 Feature package 2

Standard `Vector`

Because they are common, support of plain `Vector` fields in the data struct is desired. The strategy needs to be defined: stored as `std::vector`, stored as C-array with a branch automatically added to hold the size, offering both options with a global switch to select the option, etc.

Nested structs

Fields of a struct, which are themselves structs will be supported. The fields of type `struct` can also contain fields of this type.,

[3] The symbol hold by `N` symbol can be retrieved using a method defined as `dynvectorlen(::Type{DynVector{T,N}}) = N`

Vector of structs and Vector of Vectors

Fields which are homogeneous[4] vectors of structs or of vectors will be supported. Here "vector" stands for one of the `AbstractVector` types supported in the baseline.

Tuple

The Julia types `Tuple` (and `NTuple`) for the table rows will be supported to the same extend as structs. Nested `Tuple`: the `Tuples` can contain `structs` and `structs` can contain `Tuple`.

5 Feature package 3

We expect we can already store object of any C++ type into a `TFile` or a `TTree` of a `TFile` using a ROOT wrapper produced with WrapIt!, provided the type dictionary is included in the wrapper shared library and the type of the object to store is wrapped. Storing instances of ROOT classes derived from `TObject` and already included in `ROOT.jl` is straightforward and no extra support is needed. The `ex002-ROOT` example directory of WrapIt that contains ROOT.jl includes examples for `TH1` and for `TGraph`.

Storing instances of classes that does not inherit from `TObject` has not been tested yet. We expect it to require:

- Including in the wrapper (can be a separate shared library and module than the main ROOT wrapper), the wrapping of the class, since it will need to be instantiated from Julia;
- Including in the wrapper shared library the class dictionary generated with `rootcling` or `ACLiC`. See Custom ROOT I/O.
- Including in the wrapper the binding to `TDirectory::WriteObject-<ObjectToStoreClass>(const T*, const char*, Option_t*, Int_t)`. It can be achieved by simply adding an explicit instantiation this templated method for the type of the object to store (denoted here `ObjectToStoreClass`) in the code passed to WrapIt! to generate the wrapper.

In order to be able to store the object (including instances of a class derived from `TObject`) into a `TTree` without requiring pointer manipulation,

[4] that is, from which all elements are of the same type

the `TBranchPtr` class and `SetAddress` function defined in `ex002-ROOT/-TBranchPtr.h` must be instantiated for the object type when generating the wrapper.

This feature package consists in providing tools to ease generation of the Julia wrappers needed for custom class I/O and to provide an interface consistent with the one provided with the other feature packages.

6 Feature package 4

The ROOT framework includes a replacement of `TTree`, which is currently experimental and will eventually replace `TTree`, called RNtuple. This feature package adds support for RNtuple.

7 GSoC project

Depending on the difficulties encountered, the proposed GSoC project is expected to cover the baseline, number 1, and number 2 packages:

- **Nominal expectation** Baseline + package 1
- **Good project advancement** Baseline + packages 1 and 2
- **Minimal expectation** Baseline