UnROOT.jl Past, Now, Future

Jerry Ling[1] Tamás Gál[2]

Nov. 07, 2023

[1]Harvard Univseristy/ATLAS

[2]Erlangen Centre for Astroparticle Physics

What is UnROOT.jl

Despite the name, this package is all about ROOT:

- Parsing most popular objects from `.root` files:
  - Most notably the data (TTree, RNTuple)
- Implements `Tables.jl` interface for TTree/RNTuple
- You can write naturally fast event-loop
- Multi-threading friendly

What is UnROOT.jl

Despite the name, this package is all about ROOT:

- Parsing most popular objects from `.root` files:
  - Most notably the data (TTree, RNTuple)
- Implements `Tables.jl` interface for TTree/RNTuple
- You can write naturally fast event-loop
- Multi-threading friendly

In short, UnROOT.jl is similar to uproot in that there's no C++ dependency, but users can write both columnar style or loops directly — or compose with any of the Julia data ecosystem.

Past

After using only Julia for an ATLAS analysis, it feels a bit strange to introduce our work for the "first time."

[1] with tremendous help from Tamás and Nick Amin

Past

After using only Julia for an ATLAS analysis, it feels a bit strange to introduce our work for the "first time."

Foundation of many features in this talk were written by me[1] in a basement in Meyrin (near CERN) during COVID.

[1] with tremendous help from Tamás and Nick Amin

[2] Try not become addicted to benchmarking

Past

After using only Julia for an ATLAS analysis, it feels a bit strange to introduce our work for the "first time."

Foundation of many features in this talk were written by me[1] in a basement in Meyrin (near CERN) during COVID.

Julia has many researcher-maintained packages that are among the best in their fields. My take: don't be afraid to contribute; it's really easy.[2]

[1] with tremendous help from Tamás and Nick Amin

[2] Try not become addicted to benchmarking

Structure of the Talk

The first part is a gentle introduction to UnROOT.jl while assuming minimal Julia knowledge (~10 minutes), this will give you a flavor of the package, basic use pattern, etc.

Structure of the Talk

The first part is a gentle introduction to UnROOT.jl while assuming minimal Julia knowledge (~10 minutes), this will give you a flavor of the package, basic use pattern, etc.

Then, we will present how we achieved good performance (~10 minutes): the Julia engineering aspect of things and design choices.

Structure of the Talk

The first part is a gentle introduction to UnROOT.jl while assuming minimal Julia knowledge (~10 minutes), this will give you a flavor of the package, basic use pattern, etc.

Then, we will present how we achieved good performance (~10 minutes): the Julia engineering aspect of things and design choices.

Finally, some outlook to the near term future and some key discussion points.

(orange bar below *is* a progress bar)

Part 1

Basic Usage: Check metadata

```
julia> using UnROOT

julia> r = ROOTFile("./test/samples/NanoAODv5_sample.root")
ROOTFile with 2 entries and 21 streamers.
./test/samples/NanoAODv5_sample.root
├─ Events (TTree)
│  ├─ "run"
│  ├─ "luminosityBlock"
│  ├─ "event"
│  ├─ "⋮"
│  ├─ "L1_UnpairedBunchBptxPlus"
│  ├─ "L1_ZeroBias"
│  └─ "L1_ZeroBias_copy"
└─ untagged (TObjString)
```

Basic Usage: Load TTree

Interface may change when v1.0 lands

Use `;` to suppress displaying the entire table, which causes real I/O:

```
julia> tree = LazyTree("NanoAODv5_sample.root", "Events");

julia> names(tree)
1479-element Vector{String}:
 "HLT_QuadPFJet98_83_71_15"
 "L1_SingleJet200"
 "L1_SingleJet140er2p5_ETMHF90"
 ⋮
 "L1_SingleJet35er2p5"
 "HTXS_njets25"
 "L1_DoubleMu0er1p5_SQ"
```

Basic Usage: Load TTree with branch filter

```
julia> LazyTree("NanoAODv5_sample.root", "Events",
                ["Muon_pt", "Muon_eta," "Muon_phi", "Muon_mass"])
#or using RegEx
julia> LazyTree("NanoAODv5_sample.root", "Events", r"Muon_(pt|eta|phi|mass)$")
```

Row	Muon_phi	Muon_pt	Muon_eta	Muon_mass
	SubArray{Float3	SubArray{Float3	SubArray{Float3	SubArray{Float3
1	[]	[]	[]	[]
2	[-0.305, 0.99]	[19.9, 15.3]	[0.53, 0.229]	[0.106, 0.106]
3	[]	[]	[]	[]
4	[]	[]	[]	[]
5	[]	[]	[]	[]
6	[]	[]	[]	[]
7	[2.71, 1.37]	[22.2, 4.43]	[-1.13, 1.98]	[0.106, 0.106]
⋮	⋮	⋮	⋮	⋮
				993 rows omitted

Comment on Basic Usage

At this point, there are a few takeaways:

1. For columnar analysis, this is ~ all you need from UnROOT, `LazyTree` is a proper table with `Tables.jl` interface. (`tree.Muon_pt`, `tree[1:3, :]`)

Comment on Basic Usage

At this point, there are a few takeaways:

1. For columnar analysis, this is ~ all you need from UnROOT, `LazyTree` is a proper table with `Tables.jl` interface. (`tree.Muon_pt`, `tree[1:3, :]`)
2. Each column can be arbitrarily complex as long as it `<:AbstractVector`

[3] successor to TTree

Comment on Basic Usage

At this point, there are a few takeaways:

1. For columnar analysis, this is ~ all you need from UnROOT, `LazyTree` is a proper table with `Tables.jl` interface. (`tree.Muon_pt`, `tree[1:3, :]`)
2. Each column can be arbitrarily complex as long as it `<:AbstractVector`
3. The same code works for RNTuple[3] as well (the name "LazyTree" is not perfect)

[3] successor to TTree

Comment on Basic Usage

At this point, there are a few takeaways:

1. For columnar analysis, this is ~ all you need from UnROOT, `LazyTree` is a proper table with `Tables.jl` interface. (`tree.Muon_pt`, `tree[1:3, :]`)
2. Each column can be arbitrarily complex as long as it `<:AbstractVector`
3. The same code works for RNTuple[3] as well (the name "LazyTree" is not perfect)

```
julia> LazyTree("./RNTuple/test_ntuple_stl_containers.root", "ntuple")
 Row │ string  vector_int32  array_float      vector_vector_i  vector_string   ⋯
     │ String  Vector{Int32} StaticArraysCor  Vector{Vector{I  Vector{String} ⋯
─────┼──────────────────────────────────────────────────────────────────────────
   1 │ one     [1]           [1.0, 1.0,       Vector{Int       ["one"]         ⋯
   2 │ two     [1, 2]        [2.0, 2.0,       Vector{Int       ["one", "t      ⋯
   3 │ three   [1, 2, 3]     [3.0, 3.0,       Vector{Int       ["one", "t      ⋯
                                                               9 columns omitted
```

[3] successor to TTree

Analysis: Columnar ecosystem

While `LazyTree` is optimized for event-loop, you can use any table-compatible "sink" for columnar task. See Philippe Gras' talk for DataFrames.jl ecosystem or Ianna Osborne's talk on Awkward Array ecosystem.

```
julia> using DataFrames
julia> LazyTree("NanoAODv5_sample.root", "Events", r"Muon_"; sink=DataFrame)
1000×52 DataFrame
...

julia> using AwkwardArray
julia> LazyTree("NanoAODv5_sample.root", "Events", r"Muon_"; sink=AwkwardArray.from_table)
1000-element AwkwardArray.RecordArray
...
```

Analysis: Event-loop

The syntax is as boring as you imagined, “just” write for-loop:

```
julia> mytree = LazyTree("NanoAODv5_sample.root", "Events")

julia> Threads.@thread for evt in mytree
           evt.Muon_pt # gives you a vector
           # make analysis cuts
           # fill histograms
       end
```

That’s it, simple, boring, and *fast*!

Analysis: Event-loop Performance Spoiler

The analysis is a simplified 4-muon to Higgs (veto Z candidates) with CMS Open Data.

Benchmark Repo: github.com/Moelf/UnROOT_RDataFrame_MiniBenchmark[4]

Language	Wall Time (1 thread)	Wall Time (4 threads)
Julia	15.48 s	4.60 s
Compiled `GetEntry` Loop	19.96 s	Not impl.
Compiled RDF	24.97 s	10.23 s
PyROOT RDF	40.22 s	10.94 s

[4]All benchmarks are done on AF UChicago with EPYC 7402

Analysis: Event-loop Performance Spoiler

The analysis is a simplified 4-muon to Higgs (veto Z candidates) with CMS Open Data.

Benchmark Repo: github.com/Moelf/UnROOT_RDataFrame_MiniBenchmark[4]

Language	Wall Time (1 thread)	Wall Time (4 threads)
Julia	15.48 s	4.60 s
Compiled `GetEntry` Loop	19.96 s	Not impl.
Compiled RDF	24.97 s	10.23 s
PyROOT RDF	40.22 s	10.94 s

Relative performance is application-dependent; the takeaway is *not* "Julia is faster than C++"

[4]All benchmarks are done on AF UChicago with EPYC 7402

Part 2: Performance and Design choices

Performance 1: Who gives it to us?

Compiler, of course!

Performance 1: Who gives it to us?

Compiler, of course!

But compilers work best if they have complete type information. This is why the "peak performance" `GetEntry` C++ program is full of information for the compiler to pick the best CPU instructions:

Performance 1: Who gives it to us?

Compiler, of course!

But compilers work best if they have complete type information. This is why the "peak performance" `GetEntry` C++ program is full of information for the compiler to pick the best CPU instructions:

```
TBranch *b_nMuon = t->GetBranch("nMuon");
size_t N = dynamic_cast<TLeaf*>(b_nMuon->GetListOfLeaves()->At(0))->GetMaximum();
UInt_t nMuon; Float_t pMuon_pt[N]; Float_t pMuon_eta[N];
Float_t pMuon_phi[N]; Float_t pMuon_mass[N]; Int_t pMuon_charge[N];

TBranch *b_Muon_pt = t->GetBranch("Muon_pt");
TBranch *b_Muon_eta = t->GetBranch("Muon_eta");
TBranch *b_Muon_phi = t->GetBranch("Muon_phi");
TBranch *b_Muon_mass = t->GetBranch("Muon_mass");
TBranch *b_Muon_charge = t->GetBranch("Muon_charge");

b_nMuon->SetAddress(&nMuon);
b_Muon_pt->SetAddress(&pMuon_pt);
b_Muon_eta->SetAddress(&pMuon_eta);
b_Muon_phi->SetAddress(&pMuon_phi);
b_Muon_mass->SetAddress(&pMuon_mass);
b_Muon_charge->SetAddress(&pMuon_charge);
```

Performance 1: Where does it come from?

The same basic concept applies to any language, including Julia, say you're calculating `sum(evt.Muon_pt)` for each event.

```
julia> evt = mytree[2] # pick an interesting event
UnROOT.LazyEvent at index 2 with 4 columns
...
julia> show(evt.Muon_pt)
Float32[19.93826, 15.303187]
julia> sum(evt.Muon_pt)
35.241447f0
```

Performance 1: Where does it come from?

The same basic concept applies to any language, including Julia, say you're calculating `sum(evt.Muon_pt)` for each event.

```
julia> evt = mytree[2] # pick an interesting event
UnROOT.LazyEvent at index 2 with 4 columns
...
julia> show(evt.Muon_pt)
Float32[19.93826, 15.303187]
julia> sum(evt.Muon_pt)
35.241447f0
```

It would be best for the compiler to know exactly what's `evt.Muon_pt`, so it can use the best "sum" in assembly code.

[5] under the hood it's just a function call `getproperty()`

Performance 1: Where does it come from?

The same basic concept applies to any language, including Julia, say you're calculating `sum(evt.Muon_pt)` for each event.

```
julia> evt = mytree[2] # pick an interesting event
UnROOT.LazyEvent at index 2 with 4 columns
...
julia> show(evt.Muon_pt)
Float32[19.93826, 15.303187]
julia> sum(evt.Muon_pt)
35.241447f0
```

It would be best for the compiler to know exactly what's `evt.Muon_pt`, so it can use the best "sum" in assembly code.

In Julia, the concept of "`evt.Muon_pt`[5] has an inferable return type" is known as "type stable".

[5] under the hood it's just a function call `getproperty()`

Performance 1: What makes something type (un)stable?

Basic idea: Imagine you're the compiler and can only see "type" of things but not their "values", can you determine the output type of an operation?

Performance 1: What makes something type (un)stable?

Basic idea: Imagine you're the compiler and can only see "type" of things but not their "values", can you determine the output type of an operation?

The fact that branches of a TTree are usually not homogeneous in type means we *must* encode `"branch name"` → `"branch type"` into the *type* of `evt` itself.

Performance 1: What makes something type (un)stable?

Basic idea: Imagine you're the compiler and can only see "type" of things but not their "values", can you determine the output type of an operation?

The fact that branches of a TTree are usually not homogeneous in type means we *must* encode `"branch name" → "branch type"` into the *type* of `evt` itself.

Essentially, it must work like the built-in type `NamedTuple`:

```
julia> nt = (; Muon_pt = [1.0, 2.0], Muon_charge = [-1, -1])
julia> typeof(nt)
NamedTuple{(:Muon_pt, :Muon_charge), Tuple{Vector{Float64}, Vector{Int64}}}
#                                          ^-- first type   ^-- second type
```

Latency and Type Stability

Fully encoding all branch types puts a lot of burden on the Julia compiler, which is perceived as latency by the users. Libraries designed for columnar analysis don't want to pay the cost.

For example, `DataFrames.jl` would simply use a dictionary:

Latency and Type Stability

Fully encoding all branch types puts a lot of burden on the Julia compiler, which is perceived as latency by the users. Libraries designed for columnar analysis don't want to pay the cost.

For example, `DataFrames.jl` would simply use a dictionary:

```
julia> getfield(df, :colindex).lookup
Dict{Symbol, Int64} with 4 entries:
  :Muon_phi  ⇒ 1
  :Muon_pt   ⇒ 2
  :Muon_eta  ⇒ 3
  :Muon_mass ⇒ 4
```

A good read from the author of `DataFrames.jl` on this topic:
bkamins.github.io/julialang/2022/07/08/iteration.html.

Latency and Type Stability

This latency is very tangible. For example, the main analysis format used by CMS (`NanoAOD`) contains some ~1500 branches:

```
julia> widetree = LazyTree("NanoAODv5_sample.root", "Events")
 Row │ HLT_QuadPFJet98  L1_SingleJet200  L1_SingleJet140  Photon_hoe       L1_DoubleTa ⋯
     │ Bool             Bool             Bool             SubArray{Float3  Bool        ⋯
─────┼──────────────────────────────────────────────────────────────────────────────────
   1 │ false            false            false            [0.152, 0.0      false       ⋯
   2 │ false            false            false            [0.0676]         false       ⋯
   3 │ false            false            false            [0.205, 0.2      false       ⋯
   4 │ true             false            false            [0.266, 0.0      false       ⋯
   5 │ false            false            false            [0.0, 0.131      false       ⋯
   6 │ false            false            false            [0.0]            false       ⋯
   7 │ false            false            false            [0.426, 0.0      false       ⋯
   8 │ false            false            false            [0.346, 0.2      false       ⋯
   ⋮ │        ⋮                ⋮                ⋮                ⋮                ⋮    ⋱
                                                          1475 columns and 992 rows omitted
```

Latency and Type Stability: Free Lunch from the compiler team

```
julia> @time using UnROOT
       @time LazyTree("./test/samples/NanoAODv5_sample.root", "Events")
#1.6.7 (LTS) (2022-07-19)
  1.869491 seconds
 17.839803 seconds
```

Latency and Type Stability: Free Lunch from the compiler team

```
julia> @time using UnROOT
       @time LazyTree("./test/samples/NanoAODv5_sample.root", "Events")
#1.6.7 (LTS) (2022-07-19)
  1.869491 seconds
 17.839803 seconds
```

```
#1.9.3 (current) (2023-08-24)
  0.835918 seconds
 14.990669 seconds
```

Latency and Type Stability: Free Lunch from the compiler team

```
julia> @time using UnROOT
       @time LazyTree("./test/samples/NanoAODv5_sample.root", "Events")
#1.6.7 (LTS) (2022-07-19)
  1.869491 seconds
 17.839803 seconds
```

```
#1.9.3 (current) (2023-08-24)
  0.835918 seconds
 14.990669 seconds
```

```
#1.10-rc1 (soon™) (2023-11-03)
  0.580467 seconds
  1.363804 seconds
```

It is a much smoother experience compared to 12 months ago for naive users who try to load everything directly.

Performance 2: Lazy access of branches

Even though `LazyEvent` is as type stable as `NamedTuple`, it is not one. The primary reason is that we do not want to load every single branch from disk/cache unless needed.

Performance 2: Lazy access of branches

Even though `LazyEvent` is as type stable as `NamedTuple`, it is not one. The primary reason is that we do not want to load every single branch from disk/cache unless needed.

This means that regardless of how wide the full TTree is, you only pay for what you used:

```
julia> length(names(tree))
1479
julia> for evt in tree
           evt.nMuon # I/O starts to happen on one branch
       end
```

Performance 2: Lazy access of branches

This is made possible by using `LazyEvent` as a cursor[6] structure: it knows which row number it represents:

```
julia> evt = mytree[2]
UnROOT.LazyEvent at index 2 with 4 columns:
...
```

[6] similar to LazyRow from StructArrays.jl

Performance 2: Lazy access of branches

This is made possible by using `LazyEvent` as a cursor[6] structure: it knows which row number it represents:

```
julia> evt = mytree[2]
UnROOT.LazyEvent at index 2 with 4 columns:
...
```

When user decides to get a particular branch `evt.Muon_pt`, it's equivalent to calling:

```
mytree.Muon_pt[2]
```

[6] similar to LazyRow from StructArrays.jl

Performance 2: Lazy access of branches

This is made possible by using `LazyEvent` as a cursor[6] structure: it knows which row number it represents:

```
julia> evt = mytree[2]
UnROOT.LazyEvent at index 2 with 4 columns:
...
```

When user decides to get a particular branch `evt.Muon_pt`, it's equivalent to calling:

```
mytree.Muon_pt[2]
```

Finally, this indexing would either use or refresh the branch cache, to avoid repeated disk I/O over the same TBasket. (events are not individually readable, only a basket at a time)

[6] similar to LazyRow from StructArrays.jl

Part 3: Things Coming Soon and Discussion

Future 1: RNTuple

RNTuple is a Run 4 (HL-LHC) technology. Implementing it in Python and Julia was a good cross-check exercise, and I contributed clarifications back to the specification: (root/#11319, root/#13094, root/#11975)

Future 1: RNTuple

RNTuple is a Run 4 (HL-LHC) technology. Implementing it in Python and Julia was a good cross-check exercise, and I contributed clarifications back to the specification: (root/#11319, root/#13094, root/#11975)

The advantage is that RNTuple takes a much more systematic approach in typing schema information. Conceptually very similar to Apache Arrow (both logical and physical layout-wise):

Future 1: RNTuple is modern

Rosetta of file formats, if you're familiar with any one of them, you can get a rough idea of the other two:

RNTuple	Parquet	Arrow/Feather
field	column	field
column	–	array
cluster	row group	row group
page list	column chunk	record batch
page	page	buffer

Future 1: RNTuple

Unlike TTree, it's conceivable that C++/Python/Julia RNTuple will have complete read/write compatibility. The initial implementation (reading only) in UnROOT.jl (after uproot) took less than three days.

Future 1: RNTuple

Unlike TTree, it's conceivable that C++/Python/Julia RNTuple will have complete read/write compatibility. The initial implementation (reading only) in UnROOT.jl (after uproot) took less than three days.

On the reading RNTuple side, I have successfully followed every breaking change in specification so far (split encoding, zig-zag encoding), and I am happy to continue.

Future 1: RNTuple

For example, we can already read `PHYSLITE`[7], which is a very complex format used by the ATLAS with 664 top-level columns, with some columns look like this:

```
⇒ Vector
    ├─ :offset ⇒ Leaf{UnROOT.Index64}(col=1119)
    └─ :content ⇒ Vector
                  ├─ :offset ⇒ Leaf{UnROOT.Index64}(col=1120)
                  └─ :content ⇒ Struct
                                └─ Symbol(":_0") ⇒ Struct
                                                   ├─ :m_persKey ⇒ Leaf{UInt32}(col=1121)
                                                   └─ :m_persIndex ⇒ Leaf{UInt32}(col=1122)
```

In English: each element in this column is a vector of vector of a struct with two fields (`m_persKey, m_persIndex`).

Discussion: What's the future priority of RNTuple, maybe writing? (in general?)

[7] https://gist.github.com/Moelf/63308270b7a8143465b39f2d8fa3f98b

Future 2: Quality of Life in Analysis

Julia is excellent at producing 80% results with 20% of effort. Thus, we effortlessly benchmark well against other tools. For example, in my entry to the Analysis Description Language (ADL) benchmark (ADLBenchmark.jl), I included this table that shows the length of the function body after stripping spaces:

Query #	Julia	RDataFrame	coffea	bigquery
1	28	81	261	127
2	93	99	270	152
3	178	289	295	171
4	126	255	330	186
...	...	...	...	...

As a demonstration of Julia's flexibility.

Future 2: Quality of Life in Analysis

Over the 2023 summer, Alex Held and I supervised an IrisHEP Fellow project — Analysis Grand Challenge in Julia(LHC_AGC.jl)[8].

Atell and I realized that, although we wrote less boilerplate in Julia, it's far from perfect. Obvious wish list: declarative systematics branches, automatic histogram variations, built-in cutflow etc.

Discussion: What's a composable interface without a performance hit? What "package" should these live in?

[8] See Atell-Yehor Krasnopolski's talk

Btw: Can't help but making histogram faster

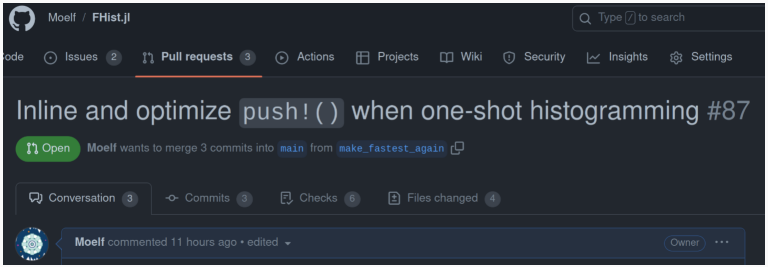

Figure 1: gotta go fast

```
julia> @benchmark Hist1D(x, range(-1,2;length=31)) setup=x=rand(10000000)
 Range (min … max):  12.546 ms …  13.519 ms  ┊ GC (min … max): 0.00% … 0.00%
```

```
> %%timeit
 h = Hist.new.Reg(30, -1, 2).Int64()
 h.fill(x)
21.1 ms ± 71.1 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
> %timeit _ = histogram1d(x, range=[-1, 2], bins=30)
13.9 ms ± 53.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Future 3: Expand Horizontally

Julia has a built-in Distributed.jl standard library, but we could use more quality of life for parallel HEP data crunching. Looking forward to the HPC Tutorial by Carsten.

Future 3: Expand Horizontally

Julia has a built-in Distributed.jl standard library, but we could use more quality of life for parallel HEP data crunching. Looking forward to the HPC Tutorial by Carsten. For the ATLAS analysis I worked on, we used HTCondor workers in real-time via ClusterManager.jl and simply `pmap()`-ed with retry:

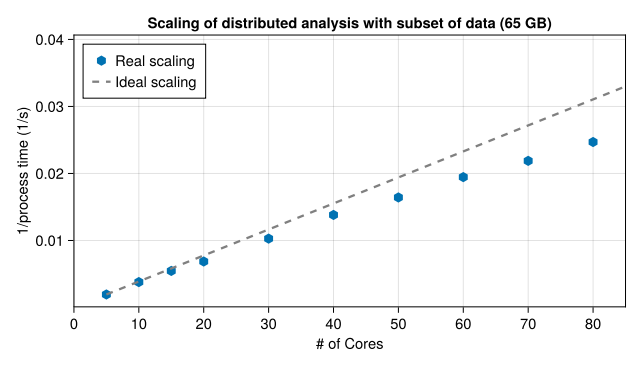

Figure 2: Scaling of naive parallelism

Future 4: Expand Up/Down Stream of Analysis

In principle, we can use Julia upstream of user analysis due to its speed and flexibility (can write any imperative code without performance degradation). In practice, unable to write `.root` files is a show stopper.[9]

[9] We used `uproot` to prepare `.root` histogram for combined fit

Future 4: Expand Up/Down Stream of Analysis

In principle, we can use Julia upstream of user analysis due to its speed and flexibility (can write any imperative code without performance degradation). In practice, unable to write `.root` files is a show stopper.[9] But maybe upstream is simply too engineering/book-keeping heavy to be worth it?

[9] We used `uproot` to prepare `.root` histogram for combined fit
[10] and gradient information from autodiff

Future 4: Expand Up/Down Stream of Analysis

In principle, we can use Julia upstream of user analysis due to its speed and flexibility (can write any imperative code without performance degradation). In practice, unable to write `.root` files is a show stopper.[9] But maybe upstream is simply too engineering/book-keeping heavy to be worth it?

The downstream direction is much brighter. The core of pyhf was implemented as a PoC in LiteHF.jl. The Julia advantage is composability: output likelihood can be used for both Frequentist fitting[10] or Bayesian inferencing (see BAT.jl by Oliver in the next step. With the progress in the HEP Statistics Serialization Standard (HS3.jl talk by Cornelius and Robin), we are likely to have a robust and interoperable statistical ecosystem in JuliaHEP.

[9] We used `uproot` to prepare `.root` histogram for combined fit

[10] and gradient information from autodiff

Discussion Questions

1. What's the most wanted/needed features? (writing? computed branch?)
2. workflow distributed computing
3. Any upstream application?
4. systematics quality of life
5. ...

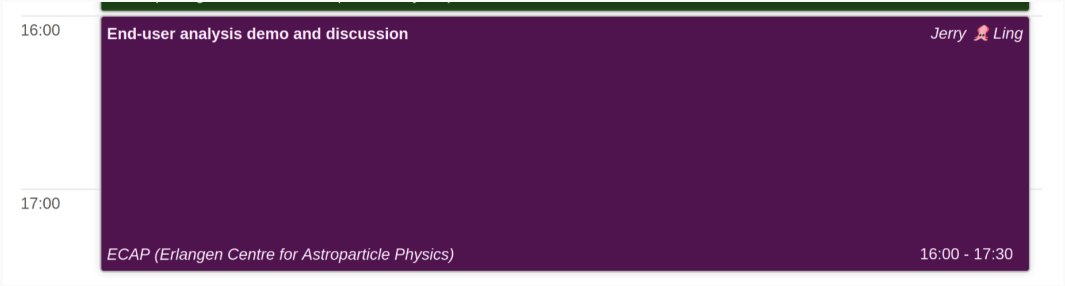

Figure 3: Looking forward to discussions