AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF PHYSICS AND APPLIED COMPUTER SCIENCE

FunRootAna

T. Bold - AGH UST, Kraków, Poland

Agenda

- Why FP?
- Features of FP
- Typical applications
- c++
- FunRootAna for plain ntuple and xAOD analyses
- Want more?

A bit of history

- In **functional** programming the focus is on telling the computer **what to do rather than how to do** it
    - FP is rooted in mathematical Category Theory - solid foundations
- Then the hardware guys came up with model closer to the hardware - **imperative** programming
    - Here you concentrate more on telling the computer of **how to** do the computation

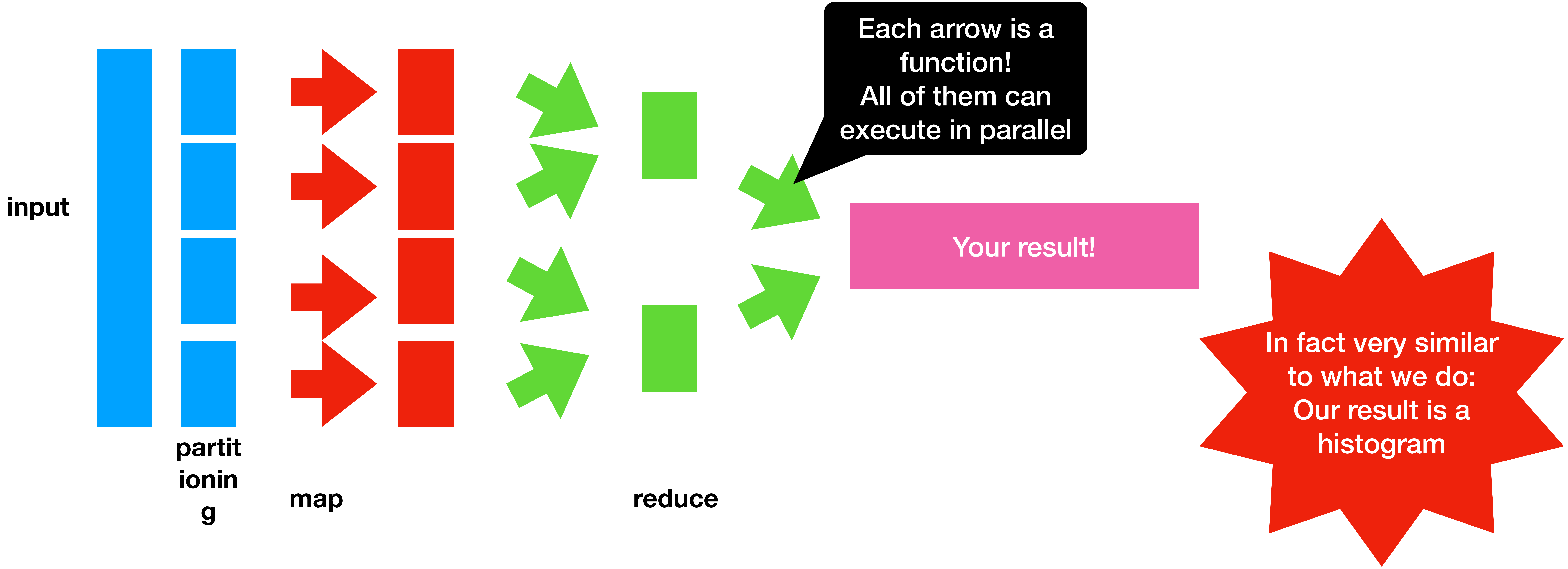
What do we write in FP?

- Functions
  - pure - w/o the side effects
  - total - always produce the result
  - higher order, partially applied, recursive …
- Functional classes
  - no modifiable state
- The profit is: a line of code does all (**and only**) what it says - referential transparency
  - Reading, understanding & maintaining such codebase is just simpler, especially in large systems
- The functional code is typically more compact - good - who likes to type!

Applications

Probably the nicest example:
Apache Spark

- Virtually everywhere but ..
- Big data through map-filter-reduce paradigm



C++

- Multi-paradigm also means functional
- Higher orders were there since long time (i.e. std::copy_if)
- First steps towards making c++ more explicit FP were made in std11 (lambdas)
    - Maybe one day shorter syntax will be available?
- std20 introduced ranges (half of the functionality, map(transform) & filter are there) - the reduce in std23
- the std::option become full fledged “maybe” Monad in std23

```
std::option(x)
      .and_then([](auto x){ return f(x);  } )
      .and_then( [](auto x){ return g(x); }) …
```

FunRootAna

https://tboldagh.github.io/FunRootAna/

- An exercise: see how far one can go with functional approach (not orthodox though) to the analysis of plain ROOT Tree and/or ATLAS xAOD
    - Did not care to match the c++ standards etc. just wanted to end up with most compact FP code to do the job
    - Did not taken care of optimisations, just tried not to do obviously bad things (e.g. runtime polymorphism)
    - And the job was to take the data-> map-filter-reduce -> histograms
- 3 ingredients:
    - a straightforward FP TTree interface,
    - functional lazy view FP container with a **complete** map-filter-reduce functionality,
    - streamlined histograms handling, +small utilities
- Can be compiled as ATLAS library or standalone ntuple analysis

Elements of FunRootAna: TTreeAccess

- TTree access class
  streamline the single branch access, via: get<type>(name)
  allow for combining branches into a custom class (i.e. TLorentzVector)
  streamline iteration over events

```
// a simple loop
 //for (PointsTreeAccess event(t); event; ++event)
 // or via functional collection interface
 TreeView<PointsTreeAccess> events(t); // the tree wrapped in an functional container

 events
 .take(2000) // take only first 2000 events
 .filter( [&](auto event){ return event.current() %2 == 1; }) // every second event (because why not)
 .foreach([&](auto event) {
// … event processing
}
```

Elements of FunRootAna: Functional container

- FunctionalInterface
  provides numerous methods (~20) to map-filter-reduce the data in containers

```
#define _rtrk(CODE) [&](const TrackInfo &_) { return CODE; }
```

```
auto loose_tracksVec = lazy_view(event.getTrackInfo())
       .filter(_rtrk(_.pt < 5.0 && _.pt > 0.3 && std::fabs(_.eta) < 2.5)).stage();

auto loose_tracks = lazy_view(loose_tracksVec);

auto tight_tracks = loose_tracks.filter(_rtrk(_.qual == TrackInfo::Tight));

const double Nch = tight_tracks.map(_rtrk(_.weight)).sum();
```

A very compact syntax to transformations.

Elements of FunRootAna: LAZY functional container

- The transformations in fact do nothing until some of the reduction step is not involved.

```
auto data = container.filter( F( std::norm(_) < 1) )
                     .filter(F(_.x>5 ))
                     .take(10)
                     .sort(F(_.x))
                     .reverse()
                     .map(F(_.y))
                     .filter(F(_>0));
auto total = data.sum();
```

No data transformation occurred here!

Operations planned above are executed here, once the sum calculation is needed.

Elements of FunRootAna: Terse syntax

- Rich API and FP approach allow expressing complex operations in a very compact & readable way

```
// data1 is a vector containing doubles
double max = std::numeric_limits<double>::min();
for ( auto el: data1) {
  if ( el > max ) {
    max = el;
  }
}
for ( auto el: data2) {
  if ( el > max ) {
    max = el;
  }
}
if ( max == std::numeric_limits<double>::min()) {
  max = 100;
}
// or with STL
double max = 100.0;
auto max_el1 = std::max_element(std::begin(data1), std::end(data1));
if ( max_el1 != std::end(data1))
  max = *max_el1;
auto max_el2 = std::max_element(std::begin(data2), std::end(data2));
if ( max_el1 != std::end(data2) && *max_el2 > *max_el1)
  max = *max_el;

// or with the functional container
const double max = data1.chain(data2)
                        .max().get().value_or(100.0);
```

This example, process two containers, and find the max in them, if no elements, then result should be 100

- Also available: rand/arithmetic/geometric streams, ranges, iota, stager, inserters.

Elements of FunRootAna: compact histograms handling

- We need to declare, book, fill and save histograms
  - That is 4 places! Typically in two files (.h and .cxx)
  - Helper class + set of macros streamline this in FunRootAna to a single line
    - placed directly in the loop over events
    - HIST1, HIST2, EFF, PROF macros to create, book, register and expose to fill operation in one go
    - set of >> operators to unify filling from PODs and lazy containers

```
int category = ..
category >> HIST1("categories_count", ";category;count of events", 5, -0.5, 4.5);
const auto x = fly::lazy_view(data); // data here is plain std::vector<float>
x >> HIST1("x", ";x[mm]", 100, 0, 100);
x >> HIST1("x_wide", ";x[mm]", 100, 0, 1000);
x.filter( F(category==0 && std::fabs(_) < 5 )) >> HIST1("x_cat_0_near_range", ";x[mm]", 100, 0, 5);
x.map( F(1./std::sqrt(_)) ) >> HIST1("sq_x", ";x[mm]^{-1/2}", 100, 0, 5);
```

Summary

- An FP approach to data analysis in ROOT is quite neat
- Typical for FP, safety, terseness, expressiveness are within the reach
- Using FunRootAna makes you to write quite a minimal amount of code (and concentrate on the essence)
- Tried in smaller and larger analyses of ATLAS data.
- The future? Up to you mostly :-).
  In few years the ROOT system will need to cooperate with c++ ranges anyway - some form of similar interface will be necessary.

Got interested?

https://tboldagh.github.io/FunRootAna/