How **You** in HEP can benefit from Functional Programming

Jonas Rembser

28.11.2018

1st LLR Student and Postdoc Seminar

Contents

- What is **Functional Programming** (FP)
- Relation between the FP and **High Energy Physics** communities
- FP in **C++** and **Python** for your **everyday work**
- FP-inspired techniques in **data analysis**
- Some easy FP paradigms to **improve your code**

Introduction

- Who of you has heard about Functional Programming before?
- Tell me what you think of this Python code:

```python
x = 4
print(x)
x = 5
print(x)
```

- Tell me what you think of this C++ code:

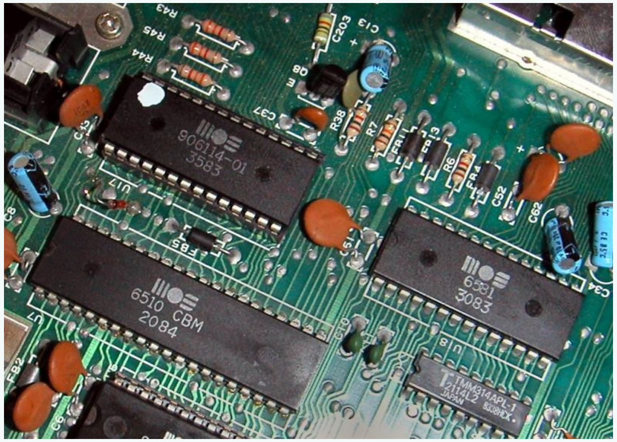
```cpp
int n = 10;

for(int i = 0; i < n; ++i)
{
    std::cout << i << " ";
}
std::cout << std::endl;
```

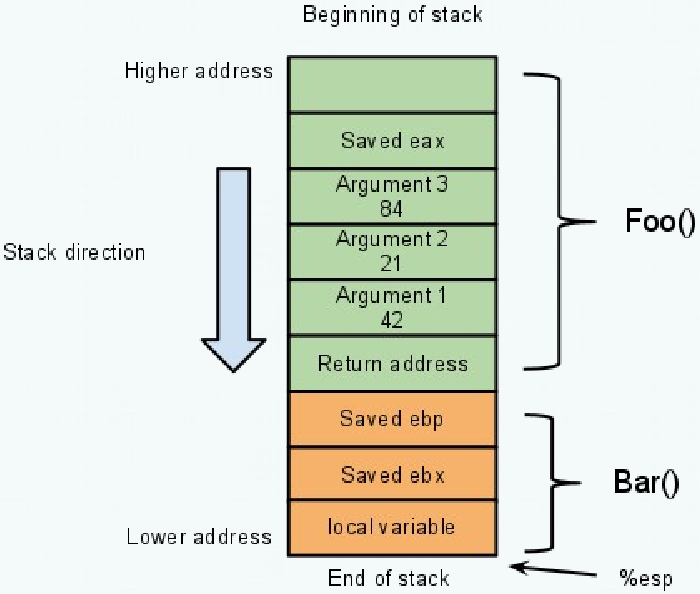
- **Lies, lies lies!** You can't trust the "definition" of variables!

But why do we lie so much?

- A historical sketch of programming:

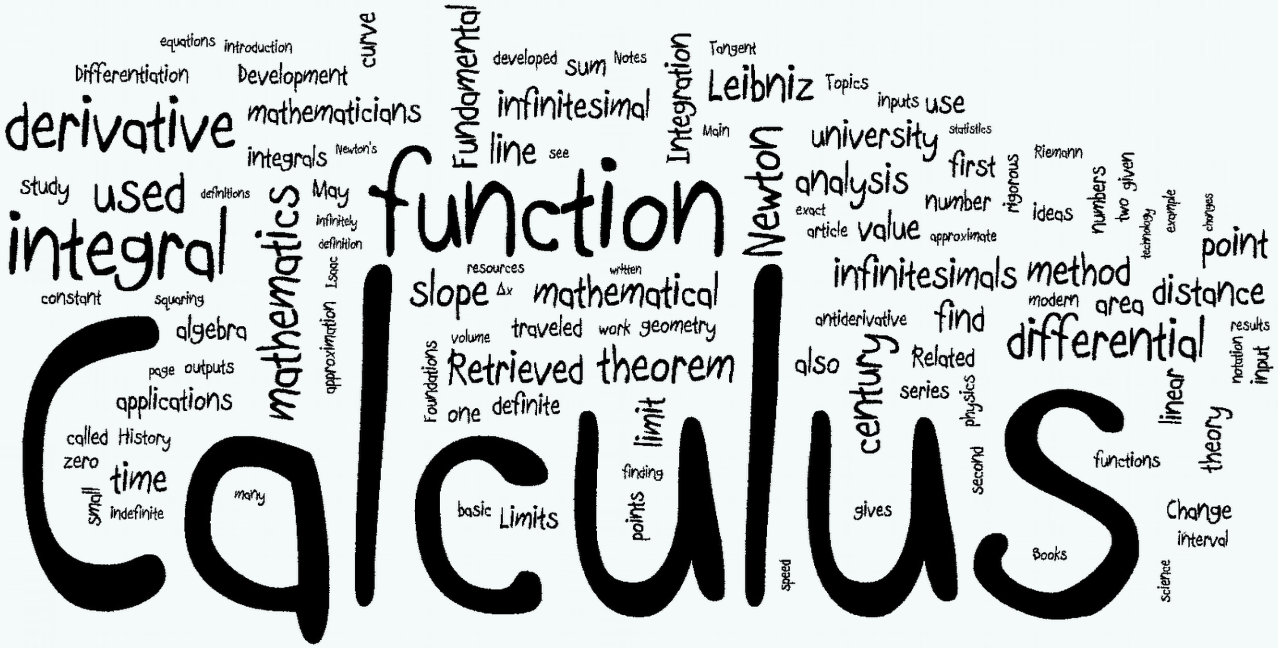
- From **machine code** to **assembly**
- From **assembly** to e.g. **Fortran** or **C**





- We got used to variables standing for a **location in memory**…
- ...not as short-hands for **values** as in mathematics!
- What is the next **level of abstraction**?

Functional Programming



- Treat computation as the **evaluation of mathematical functions**
- **Avoid** changing-state and **mutable data**
- **Higher level of abstraction** requires very smart compilers

The Lambda Calculus

- **Lambda calculus**: a formal system for expression computation based on functions as a **universal model of computation**
- Very interesting research topic that plays an important role in FP

Syntax	**Name**	**Description**
x	Variable	A character or string representing a parameter or mathematical/logical value
(λx.M)	Abstraction	Function definition (M is a lambda term). The variable x becomes bound in the expression.
(M N)	Application	Applying a function to an argument. M and N are lambda terms.
Operation	**Name**	**Description**
(λx.M[x]) → (λy.M[y])	α-conversion	Renaming the bound (formal) variables in the expression. Used to avoid name collisions.
((λx.M) E) → (M[x:=E])	β-reduction	Replacing the bound variable with the argument expression in the body of the abstraction

Why was I showing this?

- Consider trivial expression (λx.x)
- In Python it would be `lambda x: x`
- That should look familiar! **Lambda functions** are now part of many languages so you can create **unnamed** "throw-away" **functions**
- Example: `map(lambda x: x**2, [1,2,3])`
- We just saw another FP concept: Higher-order functions! `map` is a function that takes a function as an argument

Now you know why it's called "lambda function" in Python. But why "lambda calculus in the first place?

- Recursive problem… which is a good keyword

Recursion in Lambda Calculus

How to implement recursion?

- Not possible to refer to the definition of a function in a function body!

- What we need is some higher-order function to apply on a function to call it recursively:

  fix f = f (fix f), and therefore *fix f = f (f (… f (fix f) …))*.

- This fixed-point combinator can be implemented in lambda Calculus by **Haskell Curry's Y combinator**:

$$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

Why was I showing this again?

- The US company **Y Combinator** is arguably the most famous startup incubator in silicon valley
- It invested in Dropbox, Airbnb, Reddit, Docker, etc.
- If FP concepts inspire trendy company names now, **we are definitely up to something!**

Y Combinator

Functional Programming Languages

- The most famous one is **Haskell**, which sticks close to the **typed lambda calculus**. These are the strong points of the language:
  - **Purity:**
    functions have no side-effects and outputs only depends on inputs
  - **Strong typing:**
    all variable types must be known at compile-time and no implicit casting, which results in most bugs being caught at compile time and enables powerful static code analysis
  - **Elegance:**
    code often reads like a high-level algorithm description
  - **Laziness:**
    nothing is evaluated until it has to be evaluated
- In particular the first two bullet-points result in fewer bugs than you'd normally get in e.g. C++ and Python
- I will not go deeper in the topic of FP languages and will explain how to use these concepts in C++ or Python instead

Functional Programming in the Real World

- Some companies use pure FP languages like **Haskell** or **Ocaml** (the latter developed by **INRIA**)
- Mostly in silicon valley, startup scene, fin-tech sector and France

Jane Street

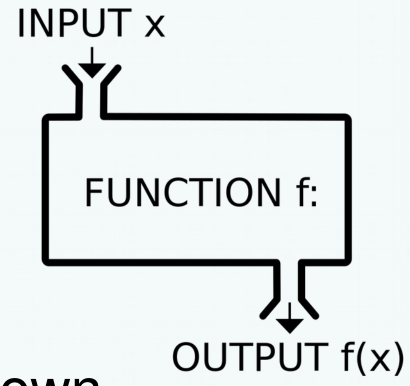
- Companies like FP languages for:
  - Robustness
  - Scalability
  - Rapid development times (once you get the hang of it)
  - Readability of code
  - It attracts high quality nerds

Functional Programming in High Energy Physics

- **Google** searches:
  - *physics+"object oriented programming"*
    2 Million results
  - *physics+"functional programing"*
    7000 results
  - **Conclusion**: we definitely didn't jump on that bandwagon…
- High energy physics has an important **Fortran legacy**
  - Fortran: as un-functional as it gets. Variables all have to be declared in the beginning and then you mutate them heavily!
  - Many people still write code like this today
- Fortunately, we use a lot of **C++** and **Python**, which now took over many concepts from PF for example:
  - Powerful **type systems** (C++)
  - **Lambda functions** and **higher-order functions** (both)
  - **Mutable variables can be avoided** without loosing much speed thanks to powerful optimizers in moders compilers (C++)

What you can take from FP to write better code **Today**

- Stick to **pure functions** as much as possible.
  - A pure functions output only depends on the input (not on some external state) and has **no "side effects"** (implicit effects on external state)
- Ensure that your variables have **well-defined** types
  - In Python you can't really do that
  - In C++ that means **avoid** `nullptr` **, casts,** `void*` and **violations** of the **Liskov Substitution principle**



- Try to find elegant, abstract solutions to your problems with **powerful higher-level functions** and define your own
  - In C++ that means get familiar with function pointers
  - In Python that means use libraries with a **high level of functional abstraction** like **Pandas** or **sklearn**
- Don't be afraid of **avoiding mutable data**
  - Don't try to mutate variables to "make your program more efficient"
  - Infest your C++ code with `const`

Making a case for **Pure Functions**

- Typical C++ HEP coding pattern on the right
- Often **huge "god classes"** which have a lot of member variables that get mutated all over the place
- That's **dangerous** because information may "leak over" to other event because you didn't reset a variable for example
- You can never be sure what the actual inputs and outputs of a function are without reading it's body
- Don't be that guy

```
class MyAnalyzer() {

    public:
        MyAnalyzer(Configuration config);

        run(Event iEvent); // for event loop

    private:

        // Some functions that do
        // something to the class members.
        // Real inputs and outputs NOT CLEAR!

        void selectElectrons(float cut);
        void makeSuperClusters(bool keepBrems);
        bool fitTrack(int nHits);

        // Metric shit-ton of variables
        // which all get mutated in
        // the class member functions
        // (hidded side effects...)
        int iEvent_;
        float pt_;
        float eta_;
        float phi_;
        // ..
        // ..
};
```

Making a case for **Type Safety**

- Very nice negative example is the ROOT TTree's `Branch` method:

```
TBranch * TTree::Branch ( const char * name,
                          void *       address,
                          const char * leaflist,
                          Int_t        bufsize = 32000 );
```

- Problem is the void pointer. Type of address is instead specified as argument, e.g.:

```
Float_t mass;
tree→Branch("mass", &mass, "mass/F");
```

- What if you later change the type of `mass` in the code from **"Float_t"** to **"Double_t"** and forget to change **"F"** to **"D"**?
  The compiler won't complain and you'll get **garbage** in your tree
- Why not using the compiler and type system as your ally to detect bugs early? Missed opportunity here!

```cpp
#include "TFile.h"
#include "TRandom.h"
#include "TFile.h"
#include "Ttree.h"

#include <vector>
#include <cmath>

class MySimulator {
  public:
    MySimulator() {
        tree_ = new TTree("tree", "tree");
        tree_->Branch("n"    , &nElectrons_, "n/i");
        tree_->Branch("eta"  , &etaVec_);
        tree_->Branch("phi"  , &phiVec_);
        tree_->Branch("pt"   , &ptVec_);

        rng_ = new TRandom(); // random number generator
    }

    ~MySimulator() {
        tree_->Write();
        delete rng_;
    }

    void simulate(int iEvent) {
        etaVec_.clear(); // don't forget to clear vectors at each event
        phiVec_.clear();
        ptVec_.clear();

        nElectrons_ = 4 + rng_->Integer(4); // how many electrons?
        for(int iElectron = 0; iElectron < nElectrons_; ++iElectron) {
            phiVec_.push_back( rng_->Uniform() * 2.*M_PI - M_PI );
            etaVec_.push_back( rng_->Uniform() * 5.0 - 2.5      );
            ptVec_ .push_back( rng_->Exp(10.) + 20.             );
        }
        tree_->Fill(); // fill the TTree with event information
    }

  private:

    TTree* tree_;
    TRandom* rng_;

    unsigned int nElectrons_; // variables for TTree branches
    std::vector<float> etaVec_;
    std::vector<float> phiVec_;
    std::vector<float> ptVec_;

};

int main() {
    int nEvents = 10000; // number of events to simulate

    TFile* file = TFile::Open("data.root", "RECREATE");

    MySimulator simulator{};

    for(int iEvent = 0; iEvent < nEvents; ++iEvent) {
        simulator.simulate(iEvent);
    }
    delete file;
}
```

C++ case study: filling a TTree

- Let's try to find the "non-FP" parts together:
  - One mutated variable per tree branch
  - Impure TTree::Fill() and TTree::Write() functions
  - Mutated TRandom
  - Vectors mutated all over the place:
    - std::vector<T>::clear()
    - std::vector<T>::push_back()

```cpp
#ifndef __S_TREE__
#define __S_TREE__

#include "TTree.h"

#include <mutex>

template<class T>
void addTreeBranch(TTree* tree, const char* name,  T& variable) {
    tree->Branch(name, &variable);
}

#define TYPES_WITH_SUFFIX \
     X(char           , "/B") X(unsigned char , "/b") \
     X(short          , "/S") X(unsigned short, "/s") \
     X(int            , "/I") X(unsigned int  , "/i") \
     X(long           , "/L") X(unsigned long , "/l") \
     X(float          , "/F") X(double        , "/D") \
     X(char *         , "/C") X(bool , "/O")

#define X(Type, suffix) \
template<> \
void addTreeBranch<Type>(TTree* tree, \
                         const char* name, Type& variable) { \
    tree->Branch(name, &variable, \
                 (std::string(name) + suffix).c_str()); \
}
TYPES_WITH_SUFFIX
#undef X

#undef TYPES_WITH_SUFFIX

template <typename T>
class STree {

  Public:

    STree(const char* name, const char* title)
        : tree_(new TTree(name, title))
    { init(); }

    void init();

    void write() const { tree_->Write(); }
    void print() const { tree_->Print(); }

    void fill(T && structure) const {
        std::lock_guard<std::mutex> lock {mutex_};
        structure_ = std::move(structure);
        tree_->Fill();
    }

  private:

    template<class S>
    void addBranch(const char* name, S foo) {
        addTreeBranch(tree_, name, structure_.*foo);
    }

    TTree * tree_;
    mutable T structure_;
    mutable std::mutex mutex_;

};

#endif
```

The STree – a wrapper around TTree

- Let's first make a function to add branches of the correct type by matching the type of the variable
- Let's create a wrapper around TTree which provides a **fill** function that **does not depend on the external state** but takes the branch variables as inputs
- To use this class, one has to define the tree structure as a struct in C++. T**ype checking of branches** is done **at compile time** and we avert bugs
- Put lock_guard in write function to make possible multithreaded filling

Defining the tree structure

```
#ifndef __EVENT_TREE__
#define __EVENT_TREE__

#include "Stree.h"

struct EventRecord {
    unsigned int       nElectrons;
    std::vector<float> etaVec;
    std::vector<float> phiVec;
    std::vector<float> ptVec;
};

template<>
void STree<EventRecord>::init() {
    addBranch("n"    , &EventRecord::nElectrons);
    addBranch("eta"  , &EventRecord::etaVec);
    addBranch("phi"  , &EventRecord::phiVec);
    addBranch("pt"   , &EventRecord::ptVec);
}

using EventTree = STree<EventRecord>;

# endif
```

- Next, we have to define the tree structure and corresponding STree<T>::init() function for this structure
  - You might of course also write a preprocessor macro that does both at once

```
#include "TFile.h"
#include "TRandom.h"
#include "EventTree.h"

#include <vector>
#include <cmath>


class MySimulator {

  public:

    MySimulator() : tree_("tree", "tree") {}
    ~MySimulator() { tree_.write(); }

    void simulate(int iEvent) const
    {
        // how many electrons?
        const unsigned int nElectrons {4 + rng_.Integer(4)};

        std::vector<float> etaVec(nElectrons);
        std::vector<float> phiVec(nElectrons);
        std::vector<float> ptVec(nElectrons);

        for (auto& x : etaVec) x = rng_.Uniform() * 5.0 - 2.5;
        for (auto& x : phiVec) x = rng_.Uniform() * 2.*M_PI - M_PI;
        for (auto& x : ptVec)  x = rng_.Exp(10.) + 20.;

        tree_.fill({ .nElectrons = nElectrons,
                     .etaVec     = etaVec,
                     .phiVec     = phiVec,
                     .ptVec      = ptVec       });
    }

  private:

    EventTree tree_;
    mutable TRandom rng_;
};

int main()
{
    const int nEvents {10000}; // number of events to simulate

    TFile file {"data.root", "RECREATE"};

    MySimulator simulator{};

    for(int iEvent = 0; iEvent < nEvents; ++iEvent)
        simulator.simulate(iEvent);
}
```

Our final FP solution

- **Impure** part now mostly **moved** out **to other files**
- **fill** function now much more **explicit** as it was before
- Code is **easier to read** now: a look at the fill function call is enough to get the general idea
- No type-related bugs possible with the STree wrapper
- On the **downside** about **10 % slower**
- Speed loss can be mitigated by simulating in **multiple threads**, which would not have been possible before

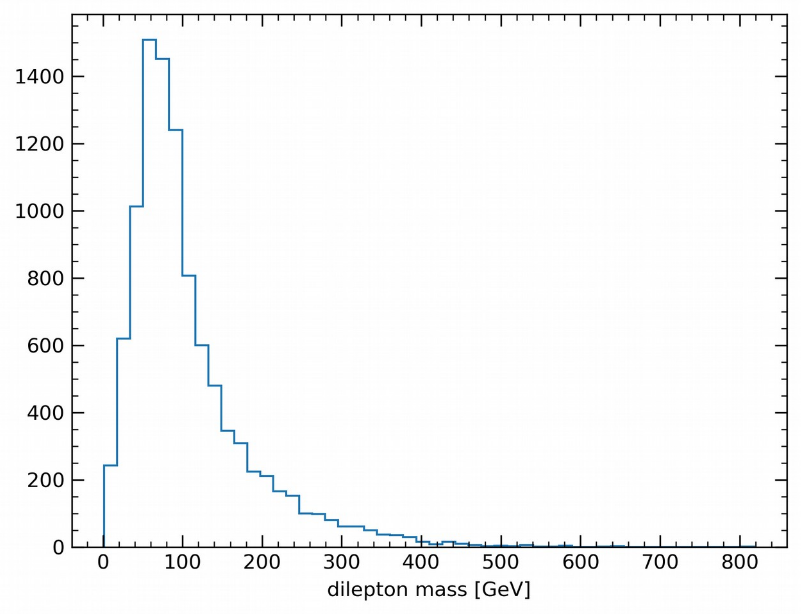
Our final FP solution (multi-threaded)

```cpp
#include "TFile.h"
#include "TRandom.h"
#include "EventTree.h"

#include <vector>
#include <cmath>


class MySimulator {

  public:

    MySimulator() : tree_("tree", "tree") {}
    ~MySimulator() { tree_.write(); }

    void simulate(int iEvent) const
    {
        // how many electrons?
        const unsigned int nElectrons {4 + rng_.Integer(4)};

        std::vector<float> etaVec(nElectrons);
        std::vector<float> phiVec(nElectrons);
        std::vector<float> ptVec(nElectrons);

        for (auto& x : etaVec) x = rng_.Uniform() * 5.0 - 2.5;
        for (auto& x : phiVec) x = rng_.Uniform() * 2.*M_PI - M_PI;
        for (auto& x : ptVec)  x = rng_.Exp(10.) + 20.;

        tree_.fill({ .nElectrons = nElectrons,
                     .etaVec     = etaVec,
                     .phiVec     = phiVec,
                     .ptVec      = ptVec       });
    }

  private:

    EventTree tree_;
    mutable TRandom rng_;
};

int main()
{
    const int nEvents {10000}; // number of events to simulate
    const int nThreads {8};

    TFile file {"data.root", "RECREATE"};

    MySimulator simulator{};

    auto simulateEvents = [&]() {
        for(int iEvent = 0; iEvent < nEvents/nThreads; ++iEvent)
            simulator.simulate(iEvent);
    };

    std::vector<std::thread> threads;
    for(int i = 0; i < nThreads; ++i) threads.emplace_back(simulateEvents);
    for(auto& thread : threads) thread.join();
}
```

- Easy to speed up your analysis by **factor n** if you kept your use of mutable data under control
- **Downside**: the order of events is now not deterministic anymore, which makes the **random numbers not deterministic**

Making Plot with Python

```
import uproot
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd


df = uproot.open("data.root")["tree"].pandas.df()


def get_mass(leptons):
    leps = leptons.sort_values("pt")[-2:]

    pt_prod  = leps["pt"].prod()
    eta_diff = leps["eta"].diff().iloc[1]
    phi_diff = leps["phi"].diff().iloc[1]

    return np.sqrt(2*pt_prod*(np.cosh(eta_diff)-np.cos(phi_diff)))


mass = df.groupby("entry").apply(get_mass).values


plt.figure()
plt.hist(mass, bins=50, histtype='step')
plt.xlabel("dilepton mass [GeV]")
plt.savefig("dilepton_mass.png")
```



- The **split-apply-combine** idiom is a nice example of how higher-oder functions can abstact away event loops
- In Pandas:
  - Split step is done by df.groupby
  - apply is the higher-order function
  - Combination step is implicit in apply
- **Pandas** provides a very high level of abstaction and is fast enough for datasets with thousands of rows
- Pandas disencourages mutable data, but you can still do it with the `inplace=True` argument

Conclusions

- Functional programming is the natural **next level of abstraction** in the evolution of programming languages
- Functional programming languages like **Haskell** or **OCaml** (while not being mainstream yet) have significantly inspired the mainstream languages to implement features originating from FP
- These new aspects of C++ and Python can help Physicists write **more bug-free and readable code**, saving time for everyone
- Three most important lessons:
  - Write **pure functions**
  - The **compiler** and **type-system** is **your friend**
  - **Avoid mutating variables** if possible
  - Use **higher-level functions**
- Sticking to these rules results in code that can be **easily parallelized**