Functional Programming
& why it’s relevant for HEP computing
About me

Doctoral student @ CERN and University of Twente (NL)

Working on ROOT, **RNTuple** to be exact

Today, I program almost exclusively in **C++**

Programming languages I’ve used before include Python, **Haskell**, C, Java, Ruby, JavaScript (& more)
Why talk about functional programming?

I like it (but am by no means an expert)

It requires a different (mental) approach to the computing problem at hand

It’s becoming more and more relevant in our modern computing landscape
→ This includes HEP computing!
Lecture outline

What is functional programming?

The essentials of functional programming

Functional thinking in the real world

Wrap-up

N.B., There will be an opportunity to see what we will discuss today in action during tomorrow’s exercise session
What is functional programming?
Declarative vs. imperative programming

Functional programming is a **declarative programming paradigm**

A declarative program describes **what** should be computed, rather than **how**

This is the opposite\(^1\) of **imperative** programs (written in e.g. C++ or Python)

---

\(^1\)Many (modern) languages draw inspiration from both paradigms, as we will see today
(Pure) functional programming

A functional program is a function which takes an input as its arguments and produces an output.

This function is defined in terms of other functions or primitives (e.g., literals).

Purely functional programs have no side effects. This means that:

- Data is immutable, the global program state cannot be altered
- The order of execution of independent items is irrelevant

In other words, functional programming separates data from behaviour.
The ingredients that give power to FP

Besides the absence of side effects, there are 4 more (interrelated) ingredients that make functional programming extremely powerful:

1. **Recursion** is considered a first-class citizen, and enables looping over data in a pure manner
2. Functions don’t have explicit return types, which allows for **partial application**
3. Functions themselves are types, which gives rise to **higher-order functions**
4. Functions are evaluated **lazily**, which means computation only happens when the result is needed
The essentials of functional programming
Haskell

Haskell is a **purely functional** language

It has several implementations, with **GHC** being the most widely used.

Programs can be **compiled** or **interpreted** interactively.

It is mostly used in **academic** settings, but also has found its way into **industry** applications from (among others) GitHub and Facebook.
A Haskell function

A Haskell function consists of two parts: the type declaration and function definition.

```
add :: Int -> Int -> Int
add x y = x + y
```
A Haskell function

A Haskell function consists of two parts: the type declaration and function definition.

```haskell
add :: Int -> Int -> Int
add x y = x + y
```
A Haskell function

A Haskell function consists of two parts: the **type declaration** and **function definition**

```
add :: Int -> Int -> Int
add x y = x + y
```
Recursion

Recursive functions are functions are defined in terms of themselves.

For example, we can compute $n!$ recursively as follows:

\[
\text{factorial} \; n = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{factorial} \; (n - 1) & \text{otherwise}
\end{cases}
\]

e.g., $\text{factorial} \; 3 = 3 \cdot \text{factorial} \; (2 \cdot \text{factorial} \; (1 \cdot \text{factorial} \; 0)) = 6$

A recursive function must have one or more base cases to prevent infinite loops!
**Recursion**

**Recursive functions** are functions defined in terms of themselves.

For example, we can compute $n!$ recursively as follows:

$$\text{factorial } n = \begin{cases} 
1 & \text{if } n = 0 \\
n \cdot \text{factorial } (n - 1) & \text{otherwise}
\end{cases}$$

e.g., $\text{factorial } 3 = 3 \cdot \text{factorial } (2 \cdot \text{factorial } (1 \cdot \text{factorial } 0)) = 6$

A recursive function must have one or more **base cases** to prevent infinite loops!

The Haskell implementation of `factorial` is left as an exercise to the reader ;)

12
Partial application

Haskell functions don’t have an explicit return type, and as a consequence can be applied partially, returning another function

\[
\text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]
\[
\text{add} \ x \ y = x + y
\]
Partial application

Haskell functions don’t have an explicit return type, and as a consequence can be applied partially, returning another function.

```
add :: Int -> Int -> Int
add x y = x + y

add_42 :: Int -> Int
add_42 x = add 42 x
```
Partial application

Haskell functions don’t have an explicit return type, and as a consequence can be applied partially, returning another function

```haskell
add :: Int -> Int -> Int
add x y = x + y

add_42 :: Int -> Int
add_42 = add 42 -- We can omit trailing arguments!
```
Partial application

Haskell functions don’t have an explicit return type, and as a consequence can be applied partially, returning another function.

\[
\text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{add } x \ y = x + y
\]

\[
\text{add}_42 :: \text{Int} \rightarrow \text{Int} \\
\text{add}_42 = \text{add} \ 42 -- \text{We can omit trailing arguments!}
\]

This allows us to build functions “on the fly”
Higher-order functions

Functions can act as types themselves, and can be provided as function arguments

\[
\text{apply}_{\text{operator}} :: \text{Num} \ a \Rightarrow (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow a \rightarrow a
\]

\[
\text{apply}_{\text{operator}} \ op \ x \ y = \text{op} \ x \ y
\]
Higher-order functions

Functions can act as types themselves, and can be provided as function arguments.

```
apply_operator :: Num a => (a -> a -> a) -> a -> a -> a
apply_operator op x y = op x y

λ> apply_operator add 9 1
10
```
Higher-order functions

Functions can act as types themselves, and can be provided as function arguments

```haskell
apply_operator :: Num a => (a -> a -> a) -> a -> a -> a
apply_operator op x y = op x y
```

```haskell
λ> apply_operator add 9 1
10
```

These functions can be defined and provided in-place with **lambda function**:

```haskell
λ> apply_operator (\ x y -> x - y) 9 1
8
```
Intermezzo: lists

A list with elements of type $\alpha$ is recursively defined as follows:

$$\text{listof } \alpha = [ ] \mid \alpha : (\text{listof } \alpha)$$

e.g., $[1, 2, 3, 4, 5] = 1 : (2 : (3 : (4 : (5 : []))))$

The first element in a list is referred to as the head, and the remaining elements as the tail.

Head $\quad$ Tail

$x = [1, 2, 3, 4, 5]$
Lazy evaluation

Lazy evaluation means the evaluation of an expression is only performed when the results are **needed by another computation**

This property, together with the previously mentioned properties, gives us powerful ways to evaluate data

```haskell
filter_odds :: [Int] -> [Int]
filter_odds = filter odd
```
Lazy evaluation

Lazy evaluation means the evaluation of an expression is only performed when the results are needed by another computation.

This property, together with the previously mentioned properties, gives us powerful ways to evaluate data.

\[
\text{filter_odds :: } [\text{Int}] \rightarrow [\text{Int}]
\]

\[
\text{filter_odds = filter \ odd}
\]

\[
\lambda > \text{filter_odds [1..5]}
\]

\[
[1, 3, 5]
\]
Lazy evaluation

Lazy evaluation means the evaluation of an expression is only performed when the results are needed by another computation.

This property, together with the previously mentioned properties, gives us powerful ways to evaluate data.

```
filter_odds :: [Int] -> [Int]
filter_odds = filter odd
```

```
λ> filter_odds [1..5]
[1, 3, 5]
```

→ What happens when we call `filter_odds [1..]`?
Lazy evaluation

Lazy evaluation means the evaluation of an expression is only performed when the results are needed by another computation.

This property, together with the previously mentioned properties, gives us powerful ways to evaluate data.

```
filter_odds :: [Int] -> [Int]
filter_odds = filter odd
```

→ What happens when we call `filter_odds [1..]`?
→ What happens when we call `take 5 (filter_odds [1..])`?
Functional thinking in the real world
Parallel, concurrent and distributed computing

Moore’s law states that the number of transistors in a microchip doubles every year (with the costs remaining constant)

Need more performance? Buy new hardware!

However, we are running into several limits:

1. The power wall: higher clock rates could lead to overheating
2. The ILP wall: a single clock cycle can only take on so many instructions at once
3. The memory wall: memory performance has lagged behind CPU performance

Instead, we have to increase performance through parallelism, concurrency and the distribution of tasks over multiple resources
Challenges in parallel programming

Parallel computing does not come for free

Two important questions to consider:

• How to make sure one task cannot alter the data used in another task?

• What if task A finishes before task B?
Challenges in parallel programming

Parallel computing does not come for free

Two important questions to consider:

- How to make sure one task cannot alter the data used in another task?
- What if task A finishes before task B?

Now, recall what we mentioned about purely functional programs and the lack of side effects.
Challenges in parallel programming

Parallel computing does not come for free

Two important questions to consider:

- How to make sure one task cannot alter the data used in another task?
  - Data is immutable, so the global program state cannot be altered
- What if task A finishes before task B?
  - The order of execution of independent items is irrelevant

Now, recall what we mentioned about purely functional programs and the lack of side effects
A pioneer in functional parallelism: MapReduce

Originally presented by Google, **MapReduce** is a programming model for **parallel** and **distributed** data processing.

It is based on two fundamental functions: **map** and **reduce**

```haskell
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

```haskell
λ> map (* 2) [1..5]
[2, 4, 6, 8, 10]
```
A pioneer in functional parallelism: MapReduce

Originally presented by Google, MapReduce is a programming model for parallel and distributed data processing.

It is based on two fundamental functions: map and reduce.

```haskell
reduce :: (a -> b -> b) -> b -> [a] -> b
reduce _ acc [] = acc
reduce f acc (x:xs) = f x (reduce f acc xs)

\> reduce (*) 1 [1..5]
120
```
MapReduce in a nutshell

1. The input data set is split and distributed over \( n \) computation units
2. A **mapper** transforms each data element into a key-value pair
3. The key-value pairs are grouped by key
4. The **reducer** merges each value belonging to a key to a single, final value
Functional patterns in other languages

Many of the concepts we’ve seen today have been adopted by imperative languages

This includes **C++** and **Python**

Other noteworthy examples include Rust, Scala and Julia

In general, languages are shifting from *single-paradigm* to *multi-paradigm*
Functional patterns in C++

C++11 introduced **lambda functions** to the language:

```cpp
auto add = [](int x, int y){ return x + y; }
```

We can use these with the **algorithms** STL library:

```cpp
std::vector<int> xs = {1, 2, 3, 4, 5};
std::vector<int> ys = {0, 0, 0, 0, 0};
std::transform(xs.begin(), xs.end(), ys.begin(),
               [](int x){ return x * 2; });
```

→ What is the value of `ys`?
Variables and function arguments are not immutable by default.

In fact, immutability can become tricky in a language that heavily relies on passing-by-reference.

Clever use of `const` qualifiers is necessary!

With these ingredients, we can start building concurrent and parallel programs using C++’s built-in `thread` library or third-party tools such as Intel’s oneTBB.
Functional patterns in Python (1)

Similar to C++, Python has the notion of lambda functions

```python
add = lambda x, y: x + y
```

Some functions, like `map` and `filter` are built in

More functions are provided with the `functools` library

```python
functools.reduce(lambda acc, x: acc * x, [1, 2, 3, 4, 5], 1)
```
In Python, lazy evaluation can be achieved with *generators*

```python
def gen_fibonacci():
a, b = 0, 1
while True:
    yield a
    a, b = b, a + b
```

```python
>>> fibonacci = gen_fibonacci()
>>> [next(fibonacci) for _ in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```
Functional thinking in HEP: RDataFrame

ROOT’s **RDataFrame** enables the creation of physics analysis using functional patterns

```cpp
ROOT::RDataFrame df("myEvents", "data.root");
auto hist =
    df.Filter("charge1 * charge2 == -1")
    .Define("invMass",
        "sqrt(2 * pt1 * pt2"
        " * (cosh(eta1 - eta2) - cos(phi1 - phi2)))")
    .Histo1D("invMass");
hist->Draw();
```
Functional thinking in HEP: RDataFrame

ROOT’s RDataFrame enables the creation of physics analysis using functional patterns.

Besides providing interfaces that resemble functional patterns, RDataFrame evaluates data lazily.

This is achieved by first creating a computation graph, and only executing this when results are requested.

By first constructing the full computation graph, parallel scheduling of the tasks becomes possible.
Wrap-up
What we’ve discussed today

Functional programming is a programming paradigm where data is separated from behaviour.
What we’ve discussed today

Functional programming is a programming paradigm where data is separated from behaviour.

This is achieved by ensuring data is immutable and the execution order of independent tasks is irrelevant.
What we’ve discussed today

Functional programming is a programming paradigm where data is separated from behaviour.

This is achieved by ensuring data is immutable and the execution order of independent tasks is irrelevant.

Other powerful features of functional programming include recursion, partial application of functions, higher-order functions and lazy evaluation.
What we’ve discussed today

Functional programming is a programming paradigm where **data** is separated from **behaviour**

This is achieved by ensuring data is **immutable** and the **execution order** of independent tasks is irrelevant

Other powerful features of functional programming include **recursion**, **partial application** of functions, **higher-order functions** and **lazy evaluation**

These ingredients give rise to patterns highly suitable for **parallel**, **distributed** and **concurrent** computing
What we’ve discussed today

Functional programming is a programming paradigm where data is separated from behaviour.

This is achieved by ensuring data is immutable and the execution order of independent tasks is irrelevant.

Other powerful features of functional programming include recursion, partial application of functions, higher-order functions and lazy evaluation.

These ingredients give rise to patterns highly suitable for parallel, distributed and concurrent computing.

Because of this, functional thinking is applied more and more outside of pure functional programming.
Exercises

Your chance to apply what we have discussed in practice!

Tomorrow (Tuesday 16/04) from 13:45-15:45 in 513/1-024

Materials can be found on Indico

No special setup needed, just your laptop and an internet connection

Come say hi :D
Further learning

For more functional programming theory:
- *Why Functional Programming Matters* (paper)
- *How Functional Programming Mattered* (paper)

For more Haskell:
- Learn You a Haskell for Great Good!
- Real World Haskell
- Monday Morning Haskell
Thank you!

& a special thanks to my mentor, Sebastien Ponce :)