RADU POPESCU, EP-SFT, CERN

PROGRAMMING LANGUAGES FOR HEP FRAMEWORKS

JOINT WLCG & HSF WORKSHOP 2018

FRAMEWORKS FOR HIGH-ENERGY PHYSICS

WHAT DOES A PHYSICS FRAMEWORK DO?

- Core processing engine for a physics collaboration
- An abstraction layer on top of computational resources
- Insulates the users from computer science aspects*
- Exploits parallel resources
- Links together many of the tools and the **developers** of a collaboration

CHARACTERISTICS

- Both computationally and IO intensive
- Event based, parallel, concurrent
- Large-scale, complex systems
- Incorporates novice programmers' algorithms
- Runs on various grid, cluster, supercomputer environments

LANGUAGE REQUIREMENTS

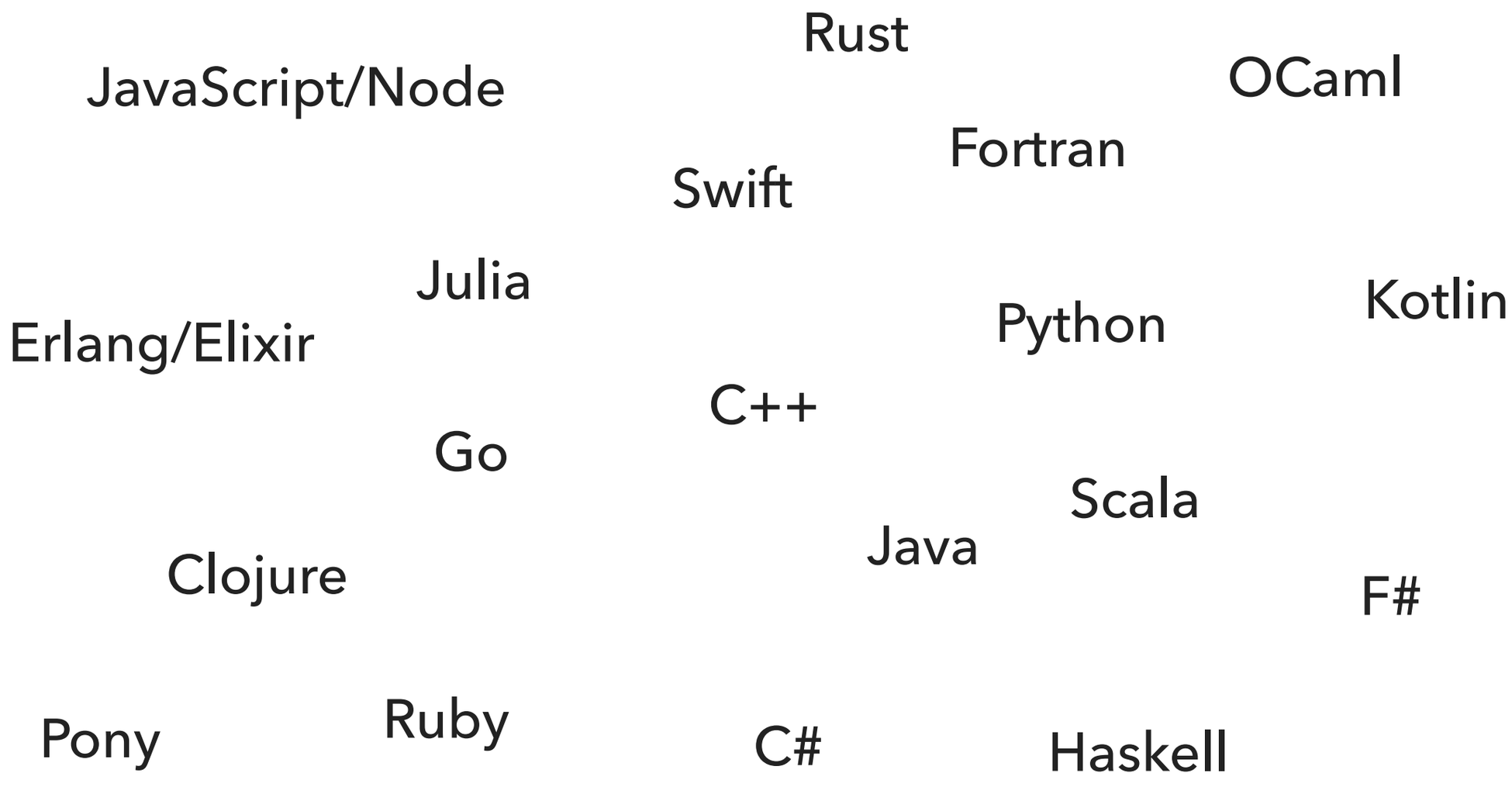
- Performance
- Parallelism
- Concurrency
- Safety
- Manage complexity
- Productivity
- Tooling

WHAT ABOUT C++?

Good	Bad
• Excellent performance • Native interoperability with OS libraries • Can implement powerful concurrency systems • Can be used for large coherent code bases	• Extremely complex, many corner cases (evolution + backward compatibility) • Memory unsafe, thread unsafe (sanitizers, checkers are opt-in, put load on CI infra) • Concurrency and optimisations create an environment where bugs can easily hide • Productivity, readability

PROGRAMMING LANGUAGES

CURRENT LANDSCAPE



TRENDS: CONCURRENCY

Support for concurrency is central to the design of modern programming languages:

- Coroutines/Green threads: Go, Haskell, Kotlin, C++20(?)
- Futures: C++(*), Rust, Scala
- Actor model: **Erlang/Elixir**, Scala (Akka), Rust (Actix), C++ (CAF)

TRENDS: TYPE SYSTEMS

- A lot of ongoing work to make advanced type system features more ergonomic
- Concurrency is fundamental and makes safety and correctness more difficult to maintain
- Leverage such features for increased safety, composition, clarity of intent
- Powerful type systems help us catch problems at compile-time, instead of at run-time

TRENDS: TYPE SYSTEMS (CONTD.)

Sum types and pattern matching

```
enum MyVariant {
  AVariantWithoutData,
  ATuple(u64, f64, String),
  AStruct { x: f64, y: f64, z: f64 },
}
```

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

```
enum Option<T> {
  Some(T),
  None,
}

let v = Option::Some(1.0);
match v {
  Some(n) => … do something with n …
  None => … nothing inside to extract …
}
```

Helps against NullPointerException

TRENDS: TYPE SYSTEMS (CONTD.)

Traits, type classes, concepts

```
fn read_exactly<R: Read>(mut rdr: R, num_bytes: usize) -> Option<Vec<u8>> {
    let mut buf = vec![0; num_bytes];
    match rdr.read(&mut buf) {
        Ok(n) if n == num_bytes => {
            Some(buf)
        }
        _ => {
            None
        }
    }
}

fn main() {
    let v = b"some binary string literal";
    let b1 = read_exactly(&v[..], 3);

    let f = File::open("/tmp/some_file.txt").unwrap();
    let b2 = read_exactly(f, 5);
}
```

TRENDS: TYPE SYSTEMS (CONTD.)

Ownership and lifetimes

- Beyond GC or reference counting
- Linear types (use value exactly once) or affine types (use value at most once) (**Rust**, Haskell)
- Borrow checking allows implementing a “mutate xor alias” system verified at compile-time (**Rust**)

TRENDS: TYPE SYSTEMS (CONTD.)

Functional(-style) programming

- Functions are values that can be manipulated as such
- Express algorithms as a series of data transformations
- Compose higher-order functions
- Minimise (shared) mutable state
- Persistent data structures - work with immutable values, algorithms have good asymptotic behaviour

TWO EXAMPLES OF LANGUAGES

Erlang	Rust
Dynamically typed	Statically typed
Simple type system	Advanced type system
Managed (BEAM)	Native (LLVM)
Distributed systems	Systems software
Fault tolerance	Safety and performance

WHAT IS ERLANG?

- Erlang is a programming language used to build massively scalable soft real-time systems with requirements on high availability, distribution and fault tolerance
- Originally used in telecoms, now sees use in banking, e-commerce, computer telephony and instant messaging.
- Created at Ericsson, open-source, open community

WHY ERLANG?

- Actor model of concurrency is at the core of Erlang's concurrency story
- Scalability - O(10^5-10^6) concurrent actors per system (see WhatsApp)
- Language VM and core frameworks are extremely "battle tested", designed for fault-tolerant systems
- Network transparency
- Many options for interoperability
- A lot of work has gone into monitoring, debugging and inspecting live systems

WHAT IS RUST?

- A modern systems programming language
- Conceived as an alternative for C++, with a type system designed for safety, without performance penalty
- Open source, open community, supported and used by Mozilla
- Stable release 2 years ago, backward compatibility is promised
- Seeing industrial adoption 100+ companies listed at: https://www.rust-lang.org/en-US/friends.html

WHY RUST?

- Predictable high performance (native, RAII, no GC)
- Advanced type system enforcing memory and thread safety
- Powerful libraries for concurrency and parallelism
- Lightweight run-time (no hidden threads)
- Low impedance bidirectional foreign function interface (FFI)
- Modern tooling (building, testing, benchmark, deployment)

CLOSING REMARKS

WHY OTHER LANGUAGES?

- Different communities have different specialties - it’s possible they have made more progress solving a problem you are facing
- Better understanding of underlying concepts - less focus on syntax, more on semantics
- Even if not adopting new language, it may be possible to backport new knowledge into existing technology stack
- The choice of language has an impact on a project: performance, safety, technical debt management, productivity

EVALUATING A NEW LANGUAGE

- As with any other technology, is a risk/reward question
- Take a piecemeal approach - identify components where risk/reward is favourable and attempt to migrate (see Firefox Quantum for C++ -> Rust)
- Successful initial attempts lower risk for future work!

THANK YOU!

BACKUP

ERLANG HIGHLIGHTS

Open Telecom Platform

- Battle-tested framework for developing distributed application
- Reusable generic components: server, finite state machine, event handler, supervisor etc.
- Extensive tooling

ERLANG HIGHLIGHTS

Actor model of concurrency

- Applications are composed of a large number of isolated actors
- Isolated memory, communication only by message passing
- Creating and destroying actors is cheap (time, memory)
- Execution of actors is multiplexed on multiple CPU cores
- This paradigm scales to a very large number of actors and helps with resiliency

ERLANG HIGHLIGHTS

Network transparency

- Same API for sending messages within node as across nodes
- All data types can be serialised/deserialised
- Easy to create multi-node applications

RUST HIGHLIGHTS

Serialization

- Serde - official framework for data serialisation and deserialisation
- Trait-based, automatic implementation for POD types
- Backends for many common formats: JSON, CBOR, Protobuf, Msgpack etc.

RUST HIGHLIGHTS

Parallelisation / vectorisation

- Full control over threading, process forking, as in C/C++
- Rayon - automatic parallelisation of iterators, backed by a work stealing scheduler on a thread pool
- SIMD - official comprehensive interface to vectorisation functionality is being stabilised

RUST HIGHLIGHTS

Concurrency

- Mio - interface to native non-blocking IO functionality
- Futures - high-performance implementation of futures and streams
- Tokio - official asynchronous IO libraries based on futures
- Actix - a full implementation of the actor model

RUST HIGHLIGHTS

Interoperability

- No overhead FFI
- Simple runtime (no hidden threads, no GC)
- Low-impedance bidirectional interaction
- Bindgen - automatically generate Rust bindings for C and C++ libraries (used by Mozilla)