PROGRAMMING CONCEPTS AND LANGUAGES, 2024W

Selected Topics: Mojo, Rust

University of Vienna

SELECTED TOPICS

Mojo Programming Language

CONTENTS

The Mojo Programming Language

Basics

- Hello World
- Functions, values, variables, etc.

Mojo Ownership System

Literature

- https://docs.modular.com/mojo/manual/basics
- https://docs.modular.com/mojo/playground

MOJO PROGRAMMING LANGUAGE

High-performance systems programming language

- Open-source standard library, may become fully open-source as it matures**
- Superset of Python
  - Makes Mojo easier to learn, while providing compatibility with Python libraries
- Strong type checking (optional)
- Memory safety
- Aiming for performance that is on par with C++ and CUDA without the complexity
- Built-in utilities for optimization, parallelization, auto-tuning

Still in development and many features are missing

- Modular* team builds on their experience building other technologies like Clang and Swift programming language
- Focus on compilation model and systems programming features (not necessarily syntax)
- Also uses lessons learned from languages like Rust, Swift, Julia, Zig, Nim, etc.

***Modular is a company founded by Chris Lattner (Swift, LLVM)**

- See https://www.modular.com
- See https://www.modular.com/blog/the-next-big-step-in-mojo-open-source

MOJO :: PERFORMANCE SHOWCASE

LANGUAGES	TIME (S)*	SPEEDUP VS PYTHON
Python 3.10.9	1027s	1X
PYPY	46.1s	22x
Scalar C++	0.20s	5,000x
Mojo🔥	0.03s	68,000x

Algorithm: **Mandelbrot**
Instance: **AWS C1.xlarge Intel Xeon**

Sources

- https://www.modular.com/mojo
- https://www.modular.com/blog/mojo-a-journey-to-68-000x-speedup-over-python-part-3

MOJO :: GOALS/HIGHLIGHTS

Full compatibility with the Python ecosystem

- Python is dominating in ML (TensorFlow, PyTorch) and Mojo also aims to address AI/ML challenges
  - Python is very productive, but anything that needs to go fast is written in C/C++

Predictable low-level performance and low-level control

Targeting heterogeneous hardware with a single language

- Ability to deploy subsets of code to accelerators
- Addressing AI/ML challenges
  - AI/ML very often used on accelerators (e.g., GPUs)
  - CPUs are equally important, not only for operations accelerators cannot handle
- Code needs to be optimized at compile time for various processing units (CPUs, GPUs, and other accelerators), which may all support different types of operations

Utilize the next-generation compiler technologies

- Built from the ground-up using **Multi-level Intermediate Representation** (**MLIR***)
  - A compiler infrastructure that's ideal for heterogeneous hardware, from CPUs and GPUs, to various AI ASIC

*we will come back to MLIR soon

some background before we move on

PYTHON, CPYTHON

Pyhton

- High-level, interpreted programming language
- Python language specification – a document that describes the Python language

CPython

- Reference implementation of Python, written in C
- Interprets Python code and compiles it into bytecode, which is executed by the CPython virtual machine
- Supports a wide range of libraries and is the most compatible with Python's ecosystem
- The compiler, standard library modules, core types, test suite, shared language specification, …

Pros

- A dominant force in AI/ML and many other fields, easy to learn, community support, packages, tooling, can be used as a “glue layer” and allows building libraries in C/C++, etc.

Cons

- Poor low-level performance, single-threaded due to global interpreter lock (GIL) in Cpython*
- Using C/C++ for performance while building libraries requires a good knowledge of internal Python and C/C++ (and perhaps CUDA and other technologies)

*There have been some improvements in this field recently

MOJO AND PYTHON

Full compatibility with Python and existing libraries

- However, it is not there yet!

Uses CPython only for interoperability

- Python 3 code without modification with full compatibility with the ecosystem (for migration and adoption purposes)
- Pure Mojo does not rely on any other runtime or compiler technology!

Mojo aims to address the need for a unified language that eliminates the need to rely on C/C++ within Python libraries

- Mojo aims for highest performance possible
- Can target different accelerators with predictability and control over how computation happens
- Uses an MLIR-based infrastructure to enable high-performance execution on a wide range of hardware

LLVM*

Started as a research project at the University of Illinois

Aim to support both static and dynamic compilation of arbitrary programming languages

LLVM Project

- A collection of modular and reusable, industrial strength, compiler and toolchain technologies
- Code in the LLVM project is licensed under the "Apache 2.0 License with LLVM exceptions”

LLVM Core

- Libraries that provide a modern source- and target-independent optimize, and code generation support for many popular CPUs
- Build around LLVM intermediate representation ("**LLVM IR**")

Clang (a frontend)

- A high-end C/C++/Objective C compiler

LLDB, libc++, compiler-rt, MLIR, OpenMP, …

*Originally stood for Low Level Virtual Machine, now the name is no longer an initialism (so just LLVM)

LLVM :: SOME INFO ON HOW COMPILERS WORK

(Clang) Frontend

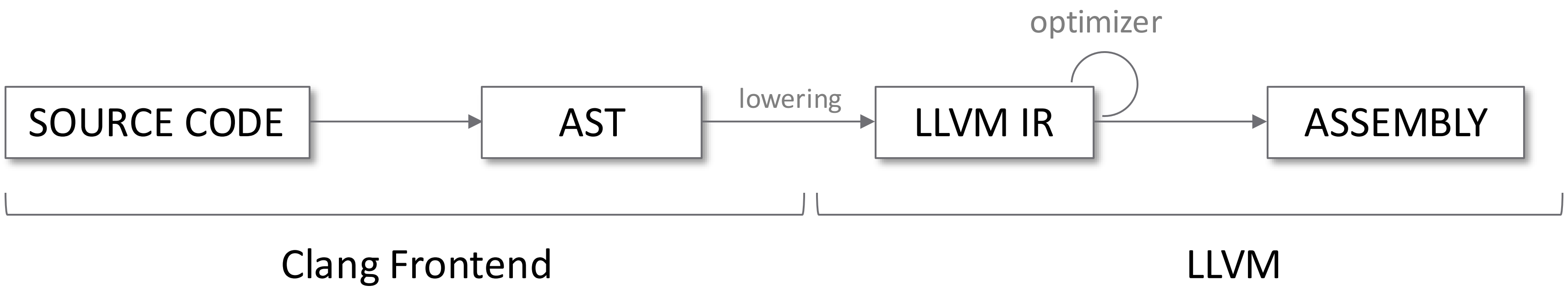
- Parses source code into an Abstract Syntax Tree (AST) and lowers it into an intermediate representation (IR)

LLVM Optimizer

- An optimizer (or “middle-end”) that transforms IR into better IR

LLVM Backend

- Converts IR into machine code for a particular platform



LLVM :: RUST EXAMPLE :: SOURCE CODE

```rust
pub fn square(x: i32) -> i32 {
    x * x
}
```

LLVM :: RUST EXAMPLE :: IR

```
@alloc_9be5c135c0f7c91e35e471f025924b11 = private unnamed_addr constant <{ [15 x i8] }> <{ [15 x i8] c"/app/example.rs" }>, align 1
@alloc_1d63fdc7829b250028d05847a3ed2432 = private unnamed_addr constant <{ ptr, [16 x i8] }> <{ ptr
@alloc_9be5c135c0f7c91e35e471f025924b11, [16 x i8] c"\0F\00\00\00\00\00\00\00\02\00\00\00\03\00\00\00" }>, align 8

define i32 @example::square::h658e3574c1514f63(i32 %x) unnamed_addr #0 !dbg !7 {
start:
%0 = call { i32, i1 } @llvm.smul.with.overflow.i32(i32 %x, i32 %x), !dbg !12
%_2.0 = extractvalue { i32, i1 } %0, 0, !dbg !12
%_2.1 = extractvalue { i32, i1 } %0, 1, !dbg !12
br i1 %_2.1, label %panic, label %bb1, !dbg !12

bb1:
ret i32 %_2.0, !dbg !13

panic:
call void @core::panicking::panic_const::panic_const_mul_overflow::h446c6be747f81624(ptr align 8 @alloc_1d63fdc7829b250028d05847a3ed2432)
#3, !dbg !12
unreachable, !dbg !12
}

declare { i32, i1 } @llvm.smul.with.overflow.i32(i32, i32) #1

declare void @core::panicking::panic_const::panic_const_mul_overflow::h446c6be747f81624(ptr align 8) unnamed_addr #2

attributes #0 = { nonlazybind uwtable "probe-stack"="inline-asm" "target-cpu"="x86-64" }
attributes #1 = { nocallback nofree nosync nounwind speculatable willreturn memory(none) }
attributes #2 = { cold noinline noreturn nonlazybind uwtable "probe-stack"="inline-asm" "target-cpu"="x86-64" }
attributes #3 = { noreturn }
```

Src: https://godbolt.org

MOJO :: MULTI-LEVEL INTERMEDIATE REPRESENTATION (MLIR)

Compiler framework (started at Google, now LLVM Developers)

Hybrid IR to support “multiple different requirements in a unified infrastructure”

- The ability to represent **dataflow graphs** (such as in TensorFlow), including dynamic shapes, the user-extensible op ecosystem, TensorFlow variables, etc.
- Optimizations and transformations typically done on such **graphs** (e.g., in Grappler)
- Ability to host **high-performance-computing-style** loop optimizations across kernels (fusion, loop interchange, tiling, etc.), and to **transform memory layouts of data**.
- Code generation “lowering” transformations such as DMA insertion, explicit cache management, memory tiling, and vectorization for 1D and 2D register architectures
- Ability to represent target-specific operations, e.g., accelerator-specific high-level operations
- Quantization and other graph transformations done on a Deep-Learning graph
- Polyhedral primitives
- Hardware Synthesis Tools / HLS

MOJO :: HELLO WORLD

Mojo requires a main() function declared with fn or def.

```
def main():
    # CHECK: Hello Mojo 🔥!
    print("Hello Mojo 🔥!")
    for x in range(9, 0, -3):
        print(x)
```

Output:

```
Hello Mojo 🔥!
9
6
3
```

How to run on your machine?

- https://docs.modular.com/mojo/manual/get-started

MOJO :: BLOCKS, STATEMENTS, COMMENTS

Blocks and statements

- Code blocks such as functions, conditions, and loops are defined with a colon followed by indented lines

```
def loop():
    for x in range(5):
        if x % 2 == 0:
            print(x)
```

Comments

- One-line comments with the hash `#` symbol
- Multi-line comments with triple quotes """ This is a comment """

```
# this is a comment

"""
This is a
comment
"""
```

MOJO :: FUNCTIONS

Two styles with minor differences

- Both styles support optional, keywords and variadic arguments

`fn` (Rust-style)

- Enforces type-checking and memory-safe behaviors
- Compile-time checks to ensure the function receives and returns the correct types
- Arguments passed by immutable reference (default)

```
fn greet2(name: String) -> String:
    return "Hello, " + name + "!"
```

`def` (Python-style)

- Allows no type declarations and dynamic behaviors
- Arguments passed by value

```
def greet(name):
    return "Hello, " + name + "!"
```

MOJO :: FUNCTIONS :: FN

`fn` (Rust-style)

- Arguments **must specify a type**
  - except for the self argument in struct methods
- Return values **must specify a type**, unless the function doesn't return a value
- If you don't specify a return type, it defaults to **None** (meaning no return value)
- By default, arguments are received as an immutable reference
  - values are read-only, using the borrowed argument convention
  - This prevents accidental mutations, and permits the use of non-copyable types as arguments
- If you want a local copy, you can simply assign the value to a local variable.
  - You can also get a mutable reference to the value by declaring the **inout argument convention**
- If the function raises an exception, it must be explicitly declared with the raises keyword
  - A def function does not need to declare exceptions

MOJO :: FUNCTIONS :: DEF

def (Python-style)

- Like in Python, but you can choose if you want to specify argument and return types
- Arguments don't require a declared type
  - Undeclared arguments are actually passed as an object, which allows the function to receive any type
    - Mojo infers the type at runtime
- Return types don't need to be declared and also default to **object***

  If a def function doesn't declare a return type of **None**, it's considered to return an **object by default**

  If an argument is an object type, it's received as a reference, following object reference semantics

  If an argument is any other declared type, it's received as a value

*The object type allows for dynamic typing because it can represent any type in the Mojo standard library, and the actual type is inferred at runtime.

```
def greet(name):
   greeting = "Hello, " + name + "!"
   return greeting
```

def (Python-like)

```
def greet(name: String) -> String:
   var greeting = "Hello, " + name + "!"
   return greeting
```

def (with types)

MOJO :: VARIABLES

A name that holds a value or object

- Mutable by default
- Constant values can be defined with the alias keyword
- Support type annotations
- Late Initialization
- Implicit type conversion

```
var a = 5
var b: Float64 = 3.14
```

Explicitly declared

```
a = 5
b = 3.14
```

Implicitly declared

Explicitly or implicitly declared

- Type is inferred (strongly typed)
  - You cannot assign a value of a different type
- Some types support implicit conversion
  - e.g., int to float
- Examples:

```
var temperature: Float64 = 99
print(temperature)
```

```
99.0
```

```
var user_id: Int = "Sam" #does not compile
count = 8 # count is type Int
count = "Nine?" # Error: can't implicitly convert 'StringLiteral' to 'Int'
```

MOJO :: VARIABLE SCOPE

Explicitly declared variables

- Lexical scoping
- Variables declared with var are bound by lexical scoping
  - Nested code blocks can read and modify variables defined in an outer scope
  - An outer scope cannot read variables defined in an inner scope at all

```
def lexical_scopes():
  var num = 1
  var dig = 1
  if num == 1:
    print("num:", num) # Reads the outer-scope "num"
    var num = 2 # Creates new inner-scope "num"
    print("num:", num) # Reads the inner-scope "num"
    dig = 2 # Updates the outer-scope "dig"
  print("num:", num) # Reads the outer-scope "num"
  print("dig:", dig) # Reads the outer-scope "dig"
```

```
num: 1
num: 2
num: 1
dig: 2
```

Implicitly declared variables

- Scoped at function level
- If the value of an implicitly-declared variable inside the if block is changed, it actually changes the value for the entire function

```
def function_scopes():
  num = 1
  if num == 1:
    print(num) # Reads the function-scope "num"
    num = 2 # Updates the function-scope variable
    print(num) # Reads the function-scope "num"
  print(num) # Reads the function-scope "num"
```

```
1
2
2
```

MOJO :: TYPES, NUMERIC TYPES

All values in Mojo have an associated data type

- Most of the types are nominal (or “named”) types, defined by a struct
- Some types that are not defined as structs
  - Functions are typed based on their signatures.
  - **NoneType** is a type with one instance, the **None** object, which is used to signal "no value”
- Basic types are included, e.g., numeric values, strings, boolean values and others
- The standard library also includes many more types that you can import as needed
  - E.g., collection types, utilities for interacting with the filesystem and getting system information, etc.

Booleans

- Either **`True`** or **`False`**
- Negated with **`not`**

Numeric Types

- Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64, UInt64, Float16, Float32, Float64
- All support number and bitwise operators, usable with the **math** module
- Numeric literals IntLiteral and FloatLiteral, which are used at compile time

MOJO :: SIMD AND DTYPE

SIMD type as a basis for numeric types (SIMD – single instruction, multiple data)

- A processor technology that allows performing an operation on an entire set of operands at once
- Fixed-size array of values that can fit into processor’s register
- Operations on SIMD values are applied elementwise, on each individual element in the vector
- A DType value defining the data type in the vector (for example, 32-bit floating-point numbers)

```
var vec1 = SIMD[DType.int8, 4](2, 3, 5, 7)
var vec2 = SIMD[DType.int8, 4](1, 2, 3, 4)

var product = vec1 * vec2

print(product)
```

```
[2, 6, 15, 28]
```

In Mojo, a scalar type is a SIMD vector with a single element

- Language design decisions, essentially all operations scalar or SIMD go through exactly the same code path

MOJO :: STRINGS

String type represents a mutable string

- Different from Python's standard string, which is immutable
- Support a variety of operators and common methods
- `StringLiteral` represents literal strings in the program source
  - Needs to be manually converted to String with `str()` method

```
var s: String = "Testing"
s += " Mojo strings"
print(s)
```

```
Testing Mojo strings
```

"Stringable" trait

- Most standard type conform to it
- Represents a type that can be converted to a string
- `str(value)` can be used for explicit conversion

```
var s = str("Items in list: ") + str(5)
print(s)
```

MOJO :: COLLECTIONS :: LIST

A dynamically-sized array of items

- Elements must be copyable and movable (**`CollectionElement`** trait)
- Element type can be passed as a parameter (`var l = List[String]()`)
- You can add, pop and access list elements

```
from collections import List

var list = List(2, 3, 5)
list.append(7)
list.append(11)
print("Popping last item: ",list.pop())
for idx in range(len(list)):
  print(list[idx], end=", ")
```

```
Popping last item: 11 2, 3, 5, 7,
```

MOJO :: COLLECTIONS :: DICT

An associative array of key-value pairs

- Key type and Value type specified as parameters
- The key type must conform to **`KeyElement`** trait
- The value elements must conform to **`CollectionElement`** trait
- **`Dict`** iterators all yield references, so you need to use the dereference operator []

```
from collections import Dict

var d = Dict[String, Float64]()
d["plasticity"] = 3.1
d["elasticity"] = 1.3
d["electricity"] = 9.7
for item in d.items():
print(item[].key, item[].value)
```

```
plasticity 3.1000000000000001
elasticity 1.3
electricity 9.6999999999999993
```

MOJO :: COLLECTIONS :: SET

An unordered collection of unique items

- Must conform to the **`KeyElement`** trait.
- Adding, removing elements and testing whether a value exists in the set
- Unions and intersections between two sets

```
from collections import Set

i_like = Set("sushi", "ice cream", "tacos", "pho")
you_like = Set("burgers", "tacos", "salad", "ice cream")
we_like = i_like.intersection(you_like)

print("We both like:")
for item in we_like:
    print("-", item[])
```

```
We both like:
- ice cream
- tacos
```

MOJO :: COLLECTIONS :: OPTIONAL

Represents a value that may or may not be present

- Evaluates to True when it holds a value, False otherwise
- If it holds a value it can be retrieved using **`value()`** method, but calling it on an **`Optional`** with no value results in an undefined behavior
  - Similar to Rust, there is an **`or_else`** method which returns the stored value if the is one, or a user-specified default value

```
var custom_greeting: Optional[String] = None
print(custom_greeting.or_else("Hello"))

custom_greeting = str("Hi")
print(custom_greeting.or_else("Hello"))
```

```
Hello
Hi
```

MOJO :: CONTROL FLOW :: IF

The `if` statement

- Short-circuit evaluation semantics
  - If the first argument to an or operator evaluates to True, the second argument is not evaluated
  - If the first argument to an and operator evaluates to False, the second argument is not evaluated

```
temp_celsius = 25
if temp_celsius > 20:
    print("It is warm.")
    print("The temperature is", temp_celsius * 9 / 5 + 32,
"Fahrenheit." )
```

```
It is warm.
The temperature is 77.0 Fahrenheit.
```

Conditional expressions

```
temp_celsius = 15
forecast = "warm" if temp_celsius > 20 else "cool"
print("The forecast for today is", forecast)
```

```
The forecast for today is cool
```

MOJO :: CONTROL FLOW :: WHILE

The `while` statement

- Repeatedly executes a code block while a given boolean expression evaluates to True
- A **`continue`** statement skips execution of the rest of the code block and resumes with the loop test expression
- A **`break`** statement terminates execution of the loop
- Can include else statement, which executes when loop's Boolean condition evaluates to False

```
n = 0
while n < 5:
  n += 1
  if n == 3:
    continue
  print(n, end=", ")
```

```
1, 2, 4, 5,
```

```
n = 0
while n < 5:
    n += 1
    if n == 3:
        break
    print(n, end=", ")
```

```
1, 2,
```

```
n = 5
while n < 4:
  print(n)
  n += 1
else:
  print("Loop completed")
```

```
Loop completed
```

MOJO :: CONTROL FLOW :: FOR

The `for` statement

- Iterates over a sequence, executing a code block for each element in the sequence
- Can iterate over any type that implements an `__iter__()` method that returns a type that defines `__next__()` and `__len__()` methods
- Can use **continue** and **break** statements
- Can use **else** clause

For example, iterating over a Mojo List

```
from collections import List

states = List[String]("California", "Hawaii", "Oregon")
for state in states:
    print(state[])
```

```
California
Hawaii
Oregon
```

MOJO :: STRUCTS

A data structure that allows you to encapsulate *fields* and *methods* that operate on an abstraction, such as a data type or an object

- Similar to a class in Python, but completely static (bound at compile-time – no runtime changes)
- Fields are variables that hold data relevant to the struct
- Methods are functions inside a struct that generally act upon the field data.

```
struct MyPair:
    var first: Int
    var second: Int

fn __init__(inout self, first: Int, second: Int):
    self.first = first
    self.second = second
```

```
var mine = MyPair(2,4)
print(mine.first)
```

Once you have a constructor, you can create an instance of MyPair and set the fields

Can have methods and static methods

- A static method can be called without creating an instance of the struct
- Does not receive the implicit "self" argument, so it cannot access any fields of the struct
  - With @staticmethod decorator
- Special methods (or dunder methods)
  - Predetermined for special tasks (e.g., `__init__()`, `__del__()`, `__copyinit__()`, `__moveinit__()`)

MOJO :: STRUCTS VS PYTHON CLASSES

Python classes are dynamic

- Allow for dynamic dispatch, monkey-patching (or “swizzling”), and dynamically binding instance fields at runtime

Mojo structs are static

- Structures and contents of a struct are set at compile time and can’t be changed while the program is running → you cannot add methods at runtime
- Structs allow you to trade flexibility for performance while being safe and easy to use
- The program knows exactly where to find the structs information and how to use it without any extra steps or delays at runtime

Mojo structs do not support inheritance ("sub-classing"), but a struct can implement traits.

- Python classes support class attributes - values that are shared by all instances of the class, equivalent to class variables or static data members in other languages
- Mojo structs don't support static data members

MOJO :: TRAITS

A trait is a set of requirements that a type must implement

- Similar to a template of characteristics for a struct
- If you want a struct with the characteristics defined in a trait, you must implement each characteristics (e.g., each method)
- Similar to Java interfaces, C++ concepts, Swift protocols, and Rust traits

You can think of it as a contract

- A type that conforms to a trait guarantees that it implements all of the features of the trait

```
trait SomeTrait:
    fn required_method(self, x: Int): ...
```

A trait

```
@value
struct SomeStruct(SomeTrait):
    fn required_method(self, x: Int):
        print("hello traits", x)
```

A struct that conforms to the trait above

```
fn fun_with_traits[T: SomeTrait](x: T):
    x.required_method(42)

fn use_trait_function():
    var thing = SomeStruct()
    fun_with_traits(thing)
```

A function that uses the trait as an argument type

MOJO :: PARAMETRIZATION

A parameter is a compile-time variable that becomes a runtime constant

- For performance reasons
- Declared in square brackets on a function or struct
- A function argument is a runtime value that is declared in parenthesis

```
fn repeat[count: Int](msg: String):
    @parameter
    for i in range(count):
        print(msg)
```

This function has

- One parameter of type **Int**
- One argument of type **String**

To call the function, you need to specify both the parameter and the argument

```
fn call_repeat():
    repeat[3]("Hello")
    # Prints "Hello" 3 times
```

Read more: https://docs.modular.com/mojo/manual/parameters/

Mojo: Value Semantics, Ownership

MOJO :: VALUE SEMANTIC

Mojo doesn't enforce value semantics or reference semantics.

- Allows each type to define how it is created, copied, and moved (if at all)
- You can choose between copy and reference semantics
- Default: value semantics, with tight control over reference semantics to avoid memory errors
- Note: Python is not value semantics

Each variable has unique access to a value, and any code outside the scope of that variable cannot modify its value

Numeric values in Mojo are value semantic because they are trivial types, which are cheap to copy.

```
x = 1
y = x
y += 1

print(x)
print(y)
```

```
1
2
```

MOJO :: OWNERSHIP

Set of rules that govern memory management

The fundamental ownership rules

- Every value has only one owner at a time
- When the lifetime of the owner ends, Mojo destroys the value
- If there are outstanding references to a value, Mojo keeps the value alive

In the future, the Mojo lifetime checker will enforce reference exclusivity, so that only one mutable reference to a value can exist at a time.

- This is not currently enforced (remember Rust?)

MOJO :: OWNERSHIP

Prevents memory safety issues such as

- Trying to free memory that has already been freed (double-free)
- Memory leaks (not freeing memory that should been freed)
- Use-after-free (accidentally deallocate data before the program is done with it)

Makes sure that there is only one “owner” for a given value at the time

- When a value’s lifetime ends, Mojo calls its destructor, which is responsible for deallocating
- When the life span of the owner ends, Mojo destroys the value
- Programmers are still responsible for making sure any type that allocates resources (including memory) also deallocates those resources in its destructor. Mojo's ownership system ensures that destructors are called promptly

Lower performance overhead

- Compared to automatic memory management (e.g., “garbage collector”)
- Relieves programmers of the burden of manual memory management and associated errors

Read more: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

MOJO :: LIFETIMES

Lifetimes

- Meaning: span of time when the value is valid
- Also refers to a specific type of parameter value used to help track the lifetimes of values and references to values

Lifetime checker

- a compiler pass that analyzes dataflow through your program and identifies when variables are valid and inserts destructor calls when a value's lifetime ends

Mojo uses lifetime values to track the validity of references. Specifically, a lifetime value answers two questions:

- What logical storage location "owns" this value?
- Can the value be mutated using this reference?

Most of the time, lifetime values are handled automatically by the compiler, but In some cases direct interaction with lifetime values is needed

- When working with references - specifically ref arguments and ref return values
- When working with types like Reference or Span which are parameterized on the lifetime of the data they refer to

MOJO :: OWNERSHIP

Mojo aims to provide full value semantics by default

- Code quality and performance is dependent upon how functions treat argument values
- Value semantics provides consistent and predictable behavior, but reference semantics is required for a systems programming language for full control over memory optimizations
- Reference semantics used in a way that ensures all code is memory safe by tracking the lifetime of every value and destroying each one at the right time (and only once)
- Achieved through the use of argument conventions that ensure every value has only one owner at a time

Argument convention

- Specifies whether an argument is mutable or immutable
- Whether the function owns the value

MOJO :: OWNERSHIP :: ARGUMENT CONVENTION

Defined by a keyword at the beginning of an argument declaration:

- **borrowed**: The function receives an immutable reference
  - The function can read the original value (it is not a copy), but it cannot mutate (modify) it. def functions treat this differently, as described below.
- **inout**: The function receives a mutable reference
  - The function can read and mutate the original value (it is not a copy)
- **owned**: The function takes ownership
  - This means the function has exclusive ownership of the argument
  - Often, this also implies that the caller should transfer ownership to this function, but that's not always what happens and this might instead be a copy (as you'll learn below)
- **ref**: The function gets a reference with an associated lifetime
  - The reference can be either mutable or immutable
  - You can think of ref arguments as a generalization of the borrowed and **inout** conventions

```
fn add(inout x: Int, borrowed y: Int):
   x += y

fn main():
    var a = 1
    var b = 2
    add(a, b)
    print(a) # Prints 3
```

MOJO :: SUMMARY

Systems programming language

Still in early development, some features are missing, for example:

- Private/public
- Extended support for traits
- Classes
- C/C++ interoperability
  - Integration to transparently import Clang C/C++ modules. Mojo's type system and C++'s are very compatible
- Calling Mojo from Python
- Full MLIR decorator reflection
- No list or dict comprehensions
  - Mojo does not yet support Python list or dictionary comprehension expressions, like [x for x in range(10)].
- No lambda syntax
- No async for or async with
  - Although Mojo has support for async functions with async fn and async def, Mojo does not yet support the async for and async with statements.
- Parameter closure captures are unsafe references
- Nested functions cannot be recursive
- … (see https://docs.modular.com/mojo/roadmap)

MOJO :: EXAMPLES

Note that Mojo provides a standard library

- https://docs.modular.com/mojo/lib

Some interesting functions

- https://docs.modular.com/mojo/stdlib/algorithm/functional/parallelize
- https://docs.modular.com/mojo/stdlib/algorithm/functional/vectorize
- https://docs.modular.com/mojo/stdlib/algorithm/functional/stencil
- https://docs.modular.com/mojo/stdlib/algorithm/functional/tile

We follow the following links:

- https://docs.modular.com/mojo/notebooks/Mandelbrot
- https://www.modular.com/blog/how-mojo-gets-a-35-000x-speedup-over-python-part-1
- https://www.modular.com/blog/how-mojo-gets-a-35-000x-speedup-over-python-part-2
- https://www.modular.com/blog/mojo-a-journey-to-68-000x-speedup-over-python-part-3

MOJO :: RESOURCES, REFERENCES AND LINKS

Resources at modular.com

1. https://www.modular.com/mojo
2. https://docs.modular.com/mojo/manual/get-started
3. https://docs.modular.com/mojo/playground
4. https://docs.modular.com/mojo/lib
5. https://docs.modular.com/mojo/notebooks/
6. https://docs.modular.com/mojo/manual/

Rust: Actix-Web

RUST :: ACTIX WEB

An HTTP server and a web framework written in Rust

- Open-source
- Type safe
- Feature rich (HTTP, HTTP2, TLS, logging, …)
- Powerful request routing
  - Can automatically extract data from incoming requests
- Support for multi-threading
- Middlewares (Logger, Session CORS)
- Developer friendly (Responders, Extractors, Form Handling, …)
- …

https://github.com/actix/actix-web

https://actix.rs, https://actix.rs/docs/

https://www.techempower.com/benchmarks/

ACTIX WEB :: HELLO WORLD

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .service(hello)
            .service(echo)
            .route("/hey", web::get().to(manual_hello))
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

async

HTTP server

Closure runs every time actix web starts a new thread

handlers

"?" -> error needs to be propagated

async function

Await for the completion

```
async fn manual_hello() -> impl Responder {
    HttpResponse::Ok().body("Hey there!")
}
```

Responder trait

HTTP Server → serving incoming requests using App as an "application factory"

App ("application factory")

- Registering routes for resources and middleware
- Storing application state across handlers withing the same scope

Cargo.toml, dependencies: actix-web = { version = "^4"}

source: https://actix.rs/docs/

ACTIX WEB :: HTTP SERVER

HttpServer is Responsible for serving HTTP requests

- Accepts an application factory as a parameter
- It must be bound to a network socket with `HttpServer::bind()`
  - For example, with `.bind(("127.0.0.1", 8080))?`
  - This can fail if the socket is in used, therefore this error must be handled
- Multi-threaded
  - `.run()` returns a Server instance
    - It has to be **awaited** or spawned
  - It automatically starts a number of workers (default: the number of physical CPUs on the system)
    - Can be altered with `.workers(4)` method
  - Each worker threads processes its requests sequentially
    - A special care should be taken when blocking a thread for any reason
      - Use asynchronous functions and futures
    - async “handlers” are executed concurrently by worker threads
  - Each thread has a **separate instance of "App" to handle requests**
  - Concurrent access to shared data must be protected (e.g., with **Arc**)

ACTIX WEB :: APPLICATION FACTORY

"App"

- Registering routes for resources and middleware
- Stores application state shared across all handlers within the same scope
- Provides an application scope that acts as a namespace for all routes

```rust
use actix_web::{web, App, HttpServer, Responder};

async fn index() -> impl Responder {
    "Hello world!"
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new().service(
        // prefixes all resources and routes attached to it...
        web::scope("/app")
        // ...so this handles requests for `GET /app/index.html`
        .route("/index.html", web::get().to(index)),
        )
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

Will match: /app, /app/

Will match: /app/index.html

source: https://actix.rs/docs/

ACTIX WEB :: APP INSTANCE, AND SHARED STATE

Shared State

- Shared with all routes and resource within the same scope
- Accessed with `web::Data<T>`
  - T is the type of the state

```rust
use actix_web::{get, web, App, HttpServer};

// This struct represents state
struct AppState {
   app_name: String,
}

#[get("/")]
async fn index(data: web::Data<AppState>) -> String {
   let app_name = &data.app_name; // <- get app_name
   format!("Hello {app_name}!")
}
// ...
```

(we still need the main)

```rust
// ...
#[actix_web::main]
async fn main() -> std::io::Result<()> {
   HttpServer::new(|| {
   App::new()
      .app_data(web::Data::new(AppState {
         app_name: String::from("Actix Web"),
      }))
      .service(index)
   })
   .bind(("127.0.0.1", 8080))?
   .run()
   .await
}
```

ACTIX WEB :: APP INSTANCE AND MUTABLE STATE

Mutable State

- `HttpServer` constructs an instance of `App` for each thread
- To share data between threads a shareable object should be used (e.g., Send + Sync*)
- Internally, web::Data uses Arc (Atomically Reference Counted**)

```rust
use actix_web::{web, App, HttpServer};
use std::sync::Mutex;

struct MyAppData {
   // Mutex for modifyinh safely across threads
   request_count: Mutex<i32>,
}


async fn index(data: web::Data<MyAppData>) -> String {
   // get MutexGuard<'_, i32>
   let mut request_count = data.request_count.lock().unwrap();
   // now we have exclusive access by a thread
   *request_count += 1;

   // We need to return string
   format!("Request count: {total_request_count}")
}
```

```rust
#[actix_web::main]
async fn main() -> std::io::Result<()> {
   // Note: web::Data created before the closure below
   let data = web::Data::new(MyAppData {
                  request_count: Mutex::new(0),
               });

   HttpServer::new(move || {
      // move the counter into the closure
      App::new()
         .app_data(data.clone()) // register the data
         .route("/", web::get().to(index))
   })
   .bind(("127.0.0.1", 8080))?
   .run()
   .await
}
```

*see: https://doc.rust-lang.org/nomicon/send-and-sync.html
**see: https://doc.rust-lang.org/std/sync/struct.Arc.html

ACTIX WEB :: CONFIGURE

Simplicity and reusability of App and `web::scope`

```
use actix_web::{web, App, HttpResponse, HttpServer};

fn scoped_config(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::resource("/test")
            .route(web::get().to(|| async { HttpResponse::Ok().body("test") }))
            .route(web::head().to(HttpResponse::MethodNotAllowed)),
    );
}

fn config(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::resource("/app")
            .route(web::get().to(|| async { HttpResponse::Ok().body("app") }))
            .route(web::head().to(HttpResponse::MethodNotAllowed)),
    );
}
```

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .configure(config)
            .service(web::scope("/api").configure(scoped_config))
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

ACTIX WEB :: EXTRACTORS

Request information access (extraction)

- Type safety
- Trait impl **`FromRequest`**
  - Many built-in extractor implementations (https://docs.rs/actix-web/latest/actix_web/trait.FromRequest.html#implementors)
- Can be accessed as an argument to a “handler” function

```
async fn index(path: web::Path<(String, String)>, json: web::Json<MyInfo>) -> impl Responder {
   let path = path.into_inner();
   format!("{} {} {} {}", path.0, path.1, json.id, json.username)
}
```

Examples

- Path, Query, Json (see next slides)
- Data (accessing pieces of application state)
- HttpRequest (parts of the request)
- String (conversion of a request payload to a String)

ACTIX WEB :: EXTRACTORS :: PATH

Path

- Information that is extracted from the request's path
- Parts of the path that are extractable are called “dynamic segments”, marked with { and }
- You can deserialize any variable segment from the path
- Usable also in combination with “**serde**” `Deserialize` trait

```
use actix_web::{get, web, App, HttpServer, Result};

/// extract path info from "/users/{user_id}/{friend}" url
/// {user_id} - deserializes to a u32
/// {friend} - deserializes to a String
#[get("/users/{user_id}/{friend}")] // <- define path parameters
async fn index(path: web::Path<(u32, String)>) -> Result<String> {
   let (user_id, friend) = path.into_inner();
   Ok(format!("Welcome {}, user_id {}!", friend, user_id))
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
   HttpServer::new(|| App::new().service(index))
             .bind(("127.0.0.1", 8080))?
             .run()
             .await
}
```

source: https://actix.rs/docs/extractors

ACTIX WEB :: EXTRACTORS :: PATH

Path

- Information that is extracted from the request's path (with `web::Path<Info>`)
- Parts of the path that are extractable are called "dynamic segments", marked with { and }
- You can deserialize any variable segment from the path
- Usable also in combination with **"serde" Deserialize trait**

```
use actix_web::{get, web, Result};
use serde::Deserialize;

#[derive(Deserialize)]
struct Info {
   user_id: u32,
   friend: String,
}

/// extract path info using serde
#[get("/users/{user_id}/{friend}")] // <- define path parameters
async fn index(info: web::Path<Info>) -> Result<String> {
   Ok(format!("Welcome {}, user_id {}!", info.friend, info.user_id ))
}

// .., main ...
```

source: https://actix.rs/docs/extractors

ACTIX WEB :: EXTRACTORS :: QUERY

Query

- With `web::Query<T>`
- Used for request parameters
- Uses serde_urlencoded behind the scenes

```
use actix_web::{get, web, App, HttpServer};
use serde::Deserialize;

#[derive(Deserialize)]
struct Info {
   username: String,
}

// this handler gets called if the query deserializes into `Info` successfully
// otherwise a 400 Bad Request error response is returned
#[get("/")]
async fn index(info: web::Query<Info>) -> String {
   format!("Welcome {}!", info.username)
}
```

source: https://actix.rs/docs/extractors

ACTIX WEB :: EXTRACTORS :: JSON

Json

- With `web::Json<Info>`
- Deserialization of request body into a struct
- Note that type T must implement the Deserialize trait

```
use actix_web::{post, web, App, HttpServer, Result};
use serde::Deserialize;

#[derive(Deserialize)]
struct Info {
    username: String,
}

/// deserialize `Info` from request's body
#[post("/submit")]
async fn submit(info: web::Json<Info>) -> Result<String> {
    Ok(format!("Welcome {}!", info.username))
}
```

source: https://actix.rs/docs/extractors

ACTIX WEB :: EXTRACTORS :: DATA

Data

- Access to application state
- With `web::Data<AppState>`
- Mutable access must be implemented

```
use actix_web::{post, web, App, HttpServer, Result};
use serde::Deserialize;

#[derive(Deserialize)]
struct Info {
   username: String,
}

/// deserialize `Info` from request's body
#[post("/submit")]
async fn submit(info: web::Json<Info>) -> Result<String> {
   Ok(format!("Welcome {}!", info.username))
}
```

source: https://actix.rs/docs/extractors

ACTIX WEB :: HANDLERS :: CUSTOM TYPES

Returning custom types is possible with the Responder trait

```rust
use actix_web::{ body::BoxBody, http::header::ContentType, HttpRequest, HttpResponse, Responder, };
use serde::Serialize;

#[derive(Serialize)]
struct MyObj {
  name: &'static str,
}

// Responder
impl Responder for MyObj {
   type Body = BoxBody;

   fn respond_to(self, _req: &HttpRequest) -> HttpResponse<Self::Body> {
      let body = serde_json::to_string(&self).unwrap();

      // Create response and set content type
      HttpResponse::Ok()
                  .content_type(ContentType::json())
                  .body(body)
      }
}

async fn index() -> impl Responder {
   MyObj { name: "user" }
}
```

source: https://actix.rs/docs/handlers

ACTIX WEB :: HANDLERS :: DIFFERENT RESPONSES

It is also possible to have a handler that returns different type of responses

- “Either” type can be used

```
use actix_web::{Either, Error, HttpResponse};

type RegisterResult = Either<HttpResponse, Result<&'static str, Error>>;

async fn index() -> RegisterResult {
    if is_a_variant() {
        // choose Left variant
        Either::Left(HttpResponse::BadRequest().body("Bad data"))
    } else {
        // choose Right variant
        Either::Right(Ok("Hello!"))
    }
}
```

source: https://actix.rs/docs/handlers

THE LAST SLIDE

Questions?