

Programming Language Concepts

Introduction

Adrien Champion
adrien.champion@email.com

1 About the class

2 Rust

From sources to runtime

- compile
- interpret

From sources to runtime

Type system

- none
- weak
- strong
- super-strong

From sources to runtime

Type system

Memory management

- manual
- heap / stack
- garbage collection

From sources to runtime

Type system

Memory management

Abstraction mechanisms

- structures
- objects
- modules
- algebraic data types
- type classes (traits)

From sources to runtime

Type system

Memory management

Abstraction mechanisms

Misc.

- compiler plugins
- macros
- indentation-has-semantics
- expressions over statements

From sources to runtime

Type system

Memory management

Abstraction mechanisms

Misc.

Software:

- (private) version control (`bitbucket` or `github`)
- `Rust` compiler (`rustc`) and project manager (`cargo`)
- unix-based system recommended (`VM` on Windows)

Software:

- (private) version control (`bitbucket` or `github`)
- `Rust` compiler (`rustc`) and project manager (`cargo`)
- unix-based system recommended (`VM` on Windows)

Resources for Rust:

official	https://www.rust-lang.org/
online compiler	https://play.rust-lang.org/
tutorial	https://doc.rust-lang.org/stable/book/ http://rustbyexample.com/
API	https://doc.rust-lang.org/stable/std/
libraries	https://crates.io/

- you **can** discuss ideas / algorithms
- you **CANNOT** share code
- I **will not** deal with cheating, the university will

- you **can** discuss ideas / algorithms
- you **CANNOT** share code
- I **will not** deal with cheating, the university will

Evaluation:

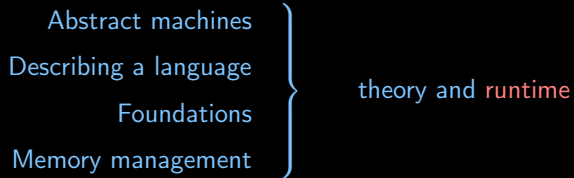
- roughly **one assignment every three weeks** in Rust
- a **midterm exam**
- a **final project** you will choose, most of the options use Rust

Abstract machines


Describing a language

Foundations

Memory management



Abstract machines
Describing a language
Foundations
Memory management



theory and runtime

Names and environment
Control structures
Control abstraction
Structuring data
Data abstraction

Abstract machines
Describing a language
Foundations
Memory management

} theory and runtime

Names and environment
Control structures
Control abstraction
Structuring data
Data abstraction

} from source to target
abstractions, control flow

Programming languages are **tools**

Depending on what the **task** is some tools are better

Programming Language Concepts help us choose the best tool for a use-case

Programming languages are **tools**

Depending on what the **task** is some tools are better

Programming Language Concepts help us choose the best tool for a use-case

Rust is **not** the ultimate tool for all tasks

but it's a **very rich** language with **recent, interesting** concepts

- **relevant** for desktop / server software
- **irrelevant** for client-side web dev

Which languages have you used or just know about?

What about them?

- in which context are they typically used? (web, software, ...)
- compiled or interpreted?
- memory management?
- type system?
- abstraction mechanisms?

① About the class

② Rust

Why?

Rust

[http://www.cnet.com/news/
samsung-joins-mozillas-quest-for-rust/](http://www.cnet.com/news/samsung-joins-mozillas-quest-for-rust/)

[http://www.cnet.com/news/
samsung-joins-mozillas-quest-for-rust/](http://www.cnet.com/news/samsung-joins-mozillas-quest-for-rust/)

- performance

compiles to LLVM,
(virtually) *no garbage collection*
zero-cost abstractions

[http://www.cnet.com/news/
samsung-joins-mozillas-quest-for-rust/](http://www.cnet.com/news/samsung-joins-mozillas-quest-for-rust/)

- performance
 - compiles to LLVM,
 - (virtually) *no garbage collection*
 - zero-cost* abstractions
- memory / thread safety
 - regional memory management*
 - strong typing, ownership, lifetimes

<http://www.cnet.com/news/samsung-joins-mozillas-quest-for-rust/>

- | | |
|--------------------------|---|
| • performance | compiles to LLVM,
(virtually) <i>no garbage collection</i>
zero-cost abstractions |
| • memory / thread safety | <i>regional memory management</i>
strong typing, ownership, lifetimes |
| • powerful abstractions | first-class functions, type classes |

<http://www.cnet.com/news/samsung-joins-mozillas-quest-for-rust/>

- | | |
|--------------------------|--|
| • performance | compiles to LLVM,
(virtually) <i>no garbage collection</i>
zero-cost abstractions |
| • memory / thread safety | <i>regional memory management</i>
strong typing, ownership, lifetimes |
| • powerful abstractions | first-class functions, type classes |
| • defensive approach | <code>Result</code> / <code>Option</code> instead of
exceptions and <code>null</code> |

Mainstream languages **lag** behind research in language theory

Most of them

- still don't provide satisfactory solutions for **concurrency**
- are **unsafe** (memory-wise, thread-wise, ...), thus **not secure**
- are rather **slow**:
virtual machine, garbage collection, ...
- force more **repetition** than they need to:
lack of powerful abstractions

Mainstream languages **lag** behind research in language theory

Most of them

- still don't provide satisfactory solutions for **concurrency**
- are **unsafe** (memory-wise, thread-wise, ...), thus **not secure**
- are rather **slow**:
virtual machine, garbage collection, ...
- force more **repetition** than they need to:
lack of powerful abstractions

Software in general is more **unsafe, slow, hard to write, read, maintain** and **extend** than it could be.

- safety and performance first
- built-in support for concurrency
- hi-level abstractions
- avoid error-prone paradigms
- encourage genericity and extensibility (and documentation)
⇒ reusability and maintainability

Life cycle of memory is always the same:

- **allocate** memory you need
- **use** it (read / write)
- **free** the memory when it is not used anymore

Generally speaking,

- **allocation** is manual, happens when declaring a variable
- **using memory** is manual: actual code

Freeing memory is a challenge, in mainstream languages it can be

- *manual*: `malloc` / `free`
 - > C (1970), C++
very old-skool
- *automatic* at runtime: garbage collection
 - > Java (1995), C#, JavaScript, Python, F#, OCaml, ...

Freeing memory is a challenge, in mainstream languages it can be

- *manual*: `malloc` / `free`
 - > C (1970), C++
 - very old-skool
- *automatic* at runtime: garbage collection
 - > Java (1995), C#, JavaScript, Python, F#, OCaml, ...
 - actually invented in 1959 for the Lisp language
 - pretty old-skool too

Garbage collection (GC):

- **easy** to use, because transparent to the developer
- memory is *automagically* freed when “not used anymore”

Garbage collection (GC):

- **easy** to use, because transparent to the developer
- memory is *automagically* freed when “not used anymore”
- **problem:** *magic* can be complex and expensive

Garbage collection (GC):

- **easy** to use, because transparent to the developer
- memory is *automagically* freed when “not used anymore”
- **problem:** *magic* can be complex and expensive

Manual memory management:

- **no overhead**, developer frees memory manually

Garbage collection (GC):

- **easy** to use, because transparent to the developer
- memory is *automagically* freed when “not used anymore”
- **problem**: *magic* can be complex and expensive

Manual memory management:

- **no overhead**, developer frees memory manually
- but very complex and thus **error-prone** (memory leaks)
- **nightmare** for concurrent threads sharing data

Garbage collection (GC):

- **easy** to use, because transparent to the developer
- memory is *automagically* freed when “not used anymore”
- **problem**: *magic* can be complex and expensive

Manual memory management:

- **no overhead**, developer frees memory manually
- but very complex and thus **error-prone** (memory leaks)
- **nightmare** for concurrent threads sharing data

In general developers ♥ garbage collection:

mindless, and fast hardware hides the overhead anyway. . .

GC shows its limits when doing **expensive computations**, potentially with a lot of allocation

For instance

- **HPC** (High Performance Calculus, applied maths),
- solving problems with **exponential complexity**,
- **web browsers** (surprisingly expensive)

Catch as many problems as possible **statically** (compile-time)

How much can we achieve with **type-checking**?

What is type-checking anyway?

What is a **type**?

A **type** usually tells

- the size of its values (*e.g.* 64 bits)
- what it represents: an integer, a pair, a struct. . .

A **type** usually tells

- the size of its values (*e.g.* 64 bits)
- what it represents: an integer, a pair, a struct. . .

With this information we can do **type-checking**:

- check that values **match** the type expected
- **failing** to compile if that's not the case, and
- **proving** “integrity” if the code type-checks

Rust tries to do more: a type is

- the usual *structural* information
- whether the “value” is *mutable*
- responsibility for *freeing* memory: *ownership*
- *lifetimes* of *references*

Rust tries to do more: a type is

- the usual *structural* information
- whether the “value” is *mutable*
- responsibility for *freeing* memory: *ownership*
- *lifetimes* of *references*

A program that type-checks is thus *proved*

- structurally *sound*
- *memory-safe* (memory leaks, dangling pointers, aliasing, ...)
- *thread-safe* (race conditions, concurrent access, ...)

Rust tries to do more: a type is

- the usual *structural* information
- whether the “value” is *mutable*
- responsibility for *freeing* memory: *ownership*
- *lifetimes* of *references*

A program that type-checks is thus *proved*

- structurally *sound*
- *memory-safe* (memory leaks, dangling pointers, aliasing, ...)
- *thread-safe* (race conditions, concurrent access, ...)

When to *deallocate* memory is a by-product of the type-checking

⇒ no overhead at runtime