Programming Language Concepts

Introduction

Adrien Champion
adrien-champion@uiowa.edu

Teaching assistant: Richard Blair
richard-blair@uiowa.edu

Contents

Schedule

Lectures every Tuesday and Thursday 2pm – 3:15pm, 106 GILH

Office hours:

Adrien Champion	Tue	3:30pm–5:30pm	*TBA*
	Thu	3:30pm–5pm	
Richard Blair	*TBA*		

Grading:

- 4 Homework Assignments (Programming Assignments) 40%
- In-class Miderm Exam 20%
- Final Project 30%
- Micro assignments 10%

Assignments / Projects

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

Assignments / Projects

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

- assignments in Rust
- a final project you will choose, most options use Rust

Resources

Piazza for asking questions about, and discussing the class:

`https://piazza.com/class/ijfcsj0kxbf10l`

Bitbucket for lectures PDFs, syllabus, projects and assignments

`https://bitbucket.org/AdrienChampion/plc`

Tools

Software:

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

Tools

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/`

Contents

Keywords

From sources to runtime

- compile
- interpret

Keywords

From sources to runtime

Type system

- none
- weak
- strong
- super-strong

Keywords

From sources to runtime

Type system

Memory management

- manual
- heap / stack
- garbage collection

Keywords

From sources to runtime

Type system

Memory management

Abstraction mechanisms

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

Keywords

From sources to runtime

Type system

Memory management

Abstraction mechanisms

Misc.

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

Keywords

From sources to runtime

Type system

Memory management

Abstraction mechanisms

Misc.

What is the point of studying PLC anyway?

- obviously useful for language designers
- what about developers?
  Many of the low level details are "invisible" to devs, why care?

What is the point of studying PLC anyway?

- obviously useful for language designers
- what about developers?
  Many of the low level details are "invisible" to devs, why care?

Programming languages are tools: different tasks call for different tools

Studying PLCs lets us

- identify quickly the best language for a given task,
- get a good idea of an unknown language quickly
  by looking at a few keywords
- write better code
  thanks to a better understanding of the *abstractions* we write it with

Roadmap

Roadmap

Abstract machines
Describing a language
Foundations
Memory management

theory and runtime

Roadmap

$$\left.\begin{array}{r}\text{Abstract machines}\\\text{Describing a language}\\\text{Foundations}\\\text{Memory management}\end{array}\right\}\quad\text{theory and runtime}$$

Names and environment

Control structures

Control abstraction

Structuring data

Data abstraction

Roadmap

$$\left.\begin{array}{r}\text{Abstract machines}\\ \text{Describing a language}\\ \text{Foundations}\\ \text{Memory management}\end{array}\right\}\quad \text{theory and runtime}$$

$$\left.\begin{array}{r}\text{Names and environment}\\ \text{Control structures}\\ \text{Control abstraction}\\ \text{Structuring data}\\ \text{Data abstraction}\end{array}\right\}\quad \begin{array}{c}\text{from source to target}\\ \text{abstractions, control flow}\end{array}$$

Your experience

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?

Contents

Why?

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

Why?

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

- performance

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

Why?

`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

Why?

`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
• powerful abstractions	first-class functions, type classes

Why?

`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
• powerful abstractions	first-class functions, type classes
• defensive approach	`Result` / `Option` instead of exceptions and `null`

Problem

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 (may or may not be a problem):
  virtual machine, *garbage collection*, ...
- force more repetition than they need to:
  lack of *powerful abstractions*

Problem

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 (may or may not be a problem):
  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.

Rust in short

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

Example: memory management

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

Memory management

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, ...

Memory management

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

Memory management

Garbage collection (GC):

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

Memory management

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

Memory management

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

Memory management

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

Memory management

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...

Problem solved?

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)

Reason: GC happens at runtime and is expensive
NB: GC is not bad, but it has a cost you should be aware of

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?

Types?

A type usually tells

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

Types?

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

More types

Rust tries to do more: a type is

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

More types

Rust tries to do more: a type is

- the usual *structural* information
- whether the "value" is mutable
- responsability 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, ...)

More types

Rust tries to do more: a type is

- the usual *structural* information
- whether the “value” is mutable
- responsability 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