Advances in Programming Languages

Lecture 18: Concurrency and More in Rust

Ian Stark

School of Informatics
The University of Edinburgh

Monday 19 November 2018
Semester 1 Week 10

THE UNIVERSITY *of* EDINBURGH

https://wp.inf.ed.ac.uk/apl18
https://course.inf.ed.ac.uk/apl

Outline

Outline

Topic: Programming for Memory Safety

The final block of lectures covers some distinctive features of the **Rust** programming language.

- Introduction: Zero-Cost Abstractions (and their cost)
- Control of Memory: Deconstructed Objects, Ownership and Borrowing
- Concurrency: Shared Memory without Data Races

Rust is a fairly new language (1.0 in 2015) that aims to support safe and efficient systems programming. In support of that aim, much of its design builds on the kinds of advances in programming languages that have appeared in this course.

Homework from Thursday

1. Do This

Work through this extremely short introduction to using Rust.

Getting Started With Rust

lil_firelord, CodinGame

https://tech.io/playgrounds/365/getting-started-with-rust

If you want to know more then I recommend: https://stevedonovan.github.io/rust-gentle-intro and then https://doc.rust-lang.org/rust-by-example and then https://doc.rust-lang.org/book

2. Watch this

RustBelt: Securing the Foundations of the Rust Programming Language

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer.

POPL 2018: 4th ACM SIGPLAN Symposium on Principles of Programming Languages

Video: **https://is.gd/rustbeltpopl2018**

Slides and more: https://plv.mpi-sws.org/rustbelt/popl18/

Week 10

Monday 19 November	Concurrency and More in Rust
Thursday 22 November	No Lecture

Week 11

Monday 26 November	Guest Lecture: Maria Gorinova
Thursday 29 November	Exam Preparation Lecture

Review and Exam Preparation !

The final lecture will be exam preparation. I'll go through some advice on reviewing course material, practising before the exam, and techniques for approaching exam questions.

Past Papers

http://wp.inf.ed.ac.uk/apl18/exam#past-papers

2007–2008

2008–2009

2009–2010

2010–2011

2014–2015

2016–2017

Next week I'll set some homework of specific past questions to work through. In the lecture itself I shall go through solutions to those questions.

Outline

Traits

Any struct or enum in Rust can have methods attached with impl, or implement a suite of several methods to match a trait, with ad-hoc polymorphism.

Trait inheritance gives subtyping, and trait bounds refine generics for parameterised structures and polymorphic functions.

All is resolved and monomorphised statically at compile time, giving strict checking of sophisticated types with no runtime overhead.

Ownership

Rust tracks *ownership* of values using *move semantics* for the handover of values in assignment, function call and return.

Move semantics lets the compiler statically check the lifetime of structured values, both on the stack and in Boxes on the heap. This guarantees memory safety without runtime overhead.

Borrowing references makes it possible to live with move semantics. Borrowing mutable references, with multiple-read single-writer, makes for C-like pointer manipulation and precise control of memory.

Object Deconstruction

Rust picks apart many of the features that go together to make objects and classes in most other languages.

- Collecting values with struct
- Alternate variants with enum
- Method implementation with impl
- Ad-hoc polymorphism with method call syntax
- Interfaces and subtyping with trait
- Heap allocation with Box
- Explicit mutability with let mut
- Call-by-reference with & and &mut

With all these we can build our own objects. As well as all kinds of other things.


BODY + LEGS + ARMS
£19
ACCESSORIES SOLD SEPARATELY


BAB


BFF
Best Friends Forever

Object Deconstruction

Rust picks apart many of the features that go together to make objects and classes in most other languages.

- Collecting values with struct
- Alternate variants with enum
- Method implementation with impl
- Ad-hoc polymorphism with method call syntax
- Interfaces and subtyping with trait
- Heap allocation with Box
- Explicit mutability with let mut
- Call-by-reference with & and &mut

With all these we can build our own objects. As well as all kinds of other things.

What Do We Win?

The key guarantee in Rust is memory safety.

- No null pointers.
- No dangling pointers.
- No reading uninitialized memory.
- No reading memory after deallocation.
- No aliasing bugs.
- No memory leaks.
- No manual deallocation.

All without reference counting, tag words, garbage collection, or other space or time overheads. Everything statically checked and assured by the compiler.

Provided you can convince the borrow checker

Trait Objects

We even now have enough to manage dynamic dispatch and runtime method selection, instead of the default static dispatch and compile-time monomorphisation.

```
trait HasDimensions {
  fn height(&self) -> i32;
  fn width(&self) -> i32;
}

fn generic_footprint<T: HasDimensions>(item: T) -> i32 {
  item.height() * item.width()
}
fn dynamic_footprint(item: &dyn HasDimensions) -> i32 {
  item.height() * item.width()
}
```

Trait Objects

Here item: &dyn HasDimensions is a *trait object* whose type is not resolved statically, but carries around a method table at runtime for *dynamic dispatch*.

```
trait HasDimensions {
  fn height(&self) -> i32;
  fn width(&self) -> i32;
}

fn generic_footprint<T: HasDimensions>(item: T) -> i32 {
  item.height() * item.width()
}
fn dynamic_footprint(item: &dyn HasDimensions) -> i32 {
  item.height() * item.width()
}
```

Trait Objects

There is an equivalent construction on the heap with trait object Box<&dyn HasDimensions>, which can contain any value implementing the HasDimensions trait.

```
trait HasDimensions {
  fn height(&self) -> i32;
  fn width(&self) -> i32;
}

fn generic_footprint<T: HasDimensions>(item: T) -> i32 {
  item.height() * item.width()
}
fn dynamic_footprint(item: &dyn HasDimensions) -> i32 {
  item.height() * item.width()
}
```

Outline

Concurrency

Rust offers concurrent programming through its standard thread library.

```
use std::thread;

fn print_hello_world () { println!("Hello, World!") }

fn main(){

let h = thread::spawn(print_hello_world);     // Pass function to be executed

let _ = h.join();                             // Wait for completion, discard result

println!("That went well");                   // Announce success

}
```

Concurrency

Rust threads can be spawned from arbitrary lambdas and closures, not just bare functions, and map directly to OS threads. Features such as thread pools, lightweight tasks, futures, promises, work stealing, data parallelism, are all under development.

Rust Win

Ownership and move semantics mean each item of data is owned by just one thread at a time.

Threads can still share access to data by borrowing references.

It's also possible for threads to borrow write access to shared store, using mutable references.

The multiple-reader single-writer constraint on borrowing mutable references means that:

Concurrent Rust programs never contain data races

Communication

Rust provides *channels* for communication between threads. The channel() function creates a new channel to carry T values and returns a pair (Sender<T>, Receiver<T>).

Communication Traits

Marker traits help manage shared resources, and are automatically inferred by the compiler.

- Send identifies types that can be safely transferred between threads.
- Sync marks types whose references can be safely shared between threads.

Sender<T> implements both Clone and Send: the capability to send on a channel can be duplicated and passed around threads.

Receiver<T> implements Send but not Clone: only one thread at a time can listen on a channel, although that ability can be transferred.

With these Rust statically guarantees multiple-producer single-consumer channel behaviour.

So Much Winning

Rust offers:

- Safe shared memory
  - No dereferencing errors
  - No aliasing bugs
  - No memory leaks
  - No manual deallocation
- Safe concurrency
  - Shared memory access
  - No data races
  - Value-passing channels
  - No channel input races

To do this uses just: strict static typing, default immutability, deconstructed objects, traits, trait bounds, ownership, reference borrowing, lifetime polymorphism, markers traits, a separate unsafe language, ...

Outline

Rust: Friends Don't Let Friends Skip Leg Day

Maybe Not So Simple: Reference Counting

Even with Box, &T and &mut T it can be hard to build and manipulate complex linked datastructures. That's why Rust provides the following, too.

- Rc<T> is a *reference counted* pointer: a runtime counter tracks how many instances of the reference still exist. It can help when there's no certainty which reference will be the last one to be used.

  Cycles of these will not be deallocated and may leak memory. Rc<T> is not safe to share between threads.

- Weak<T> is a *weak* pointer variant of Rc<T>: it will not on its own cause a value to be retained, but if the value is still there it can be accessed.

- Arc<T> is a version of Rc<T> that can be used across multiple threads, as it has an atomic reference count.

Maybe Not So Simple: Interior Mutability

Strict separation between mutable and immutable can be limiting; especially as struct fields cannot be individually mutable. So Rust has some more datatypes to help.

- Cell<T> provides *interior mutability*: the contents can be changed with get and set methods, but it's not regarded by the compiler as a mutable reference.

  Type T must have copy semantics. There is no runtime cost, but Cell<T> can give aliasing errors and invalidate datastructure assumptions and invariants.

- RefCell<T> provides interior mutability for arbitrary T. Access control is enforced with explicit borrow and borrow_mut functions that use runtime counters and may fail.

- Mutex<T> provides threadsafe interior mutability: access requires explicit request for a lock. There's no explicit release, as Rust drops this when the lock goes out of scope.

- RwLock<T> provides threadsafe interior mutability for multiple readers or a single writer, again requiring explicit lock requests.

Maybe Not So Simple: Unsafe

There's also the Unsafe Rust language, in which many of these previous structs are written.

That provides *raw pointers* *const T and *mut T which can alias, be null, and generally fail to provide the guarantees of Safe Rust.

Managing the interface between safe and unsafe is tricky too, as each can only rely on certain things from the other.

To find out more, you want this:

The Rustonomicon

The Dark Arts of Advanced and Unsafe Rust Programming

https://doc.rust-lang.org/nomicon

Outline

Summary

Safe Shared Memory

- Static checking of ownership and lifetimes, *via* default immutability and move semantics.
- Borrowing to allow multiple-reader single-writer behaviour.
- Guarantees no pointer errors or aliasing bugs, no runtime overhead.

Safe Concurrency

- Ownership tracking guarantees data-race freedom.
- Safe shared memory, as well as channel-based communication.
- Splitting Sender<T> from Receiver<T> and using marker traits ensures multiple-producer single-consumer message handling.

But at what cost? Complexity, multiplication of concerns, fighting the borrow checker, and skipping leg day?

Week 10

Monday 19 November	Concurrency and More in Rust
Thursday 22 November	No Lecture

Week 11

Monday 26 November	Guest Lecture: Maria Gorinova
Thursday 29 November	Exam Preparation Lecture

References and Borrowing

Deconstructed Burger

Things Organized Neatly, Austin Radcliffe on Tumblr

Bear and related images

Build-a-Bear Workshop

Bling Frog

Partridge & Partridge, 2016

Never Skip Leg Day

Skipping Leg Day on Know Your Meme.