Advances in Programming Languages

Lecture 18: Concurrency and More in Rust

Ian Stark

School of Informatics
The University of Edinburgh

Friday 24 November 2016
Semester 1 Week 10

https://blog.inf.ed.ac.uk/apl16

PL News (Maybe): Schiaparelli Landing

23 November 2016

The European Space Agency have reported initial analysis of extensive telemetry data transmitted during the descent of the ExoMars Schiaparelli lander.

It appears that Inertial Measurement Unit reached its maximum output measurement for a full second, longer than would be expected.

"When merged into the navigation system, the erroneous information generated an estimated altitude that was negative — that is, below ground level . . . "

" . . . In reality, the vehicle was still at an altitude of around 3.7km"

Topic: Programming for Memory Safety

The final block of lectures cover features used in the **Rust** programming language.

- Introduction: Zero-Cost Abstractions (and their cost)
- Control of Memory: Ownership
- Concurrency and more

This section of the course is entirely new — Rust itself is not that old — and I apologise for any consequent lack of polish.

The Rust Programming Language

The **Rust** language is intended as a tool for *safe systems programming*. Three key objectives contribute to this.

- Zero-cost abstractions
- Memory safety
- Safe concurrency

Basic References

https://www.rust-lang.org
https://blog.rust-lang.org

The "systems programming" motivation resonates with that for imperative C/C++. The "safe" part draws extensively on techniques developed for functional Haskell and OCaml.

Sometimes these align more closely than you might expect, often by overlap between two aims:

- Precise control for the programmer;
- Precise information for the compiler.

Previous Homework

Watch This

Peter O'Hearn: Reasoning with Big Code
https://is.gd/reasoning_big_code

Talk at the *Alan Turing Institute* about how Facebook is using automatic verification at scale to check code and give feedback to programmers.

THE ALAN TURING INSTITUTE
Computer Science
Reasoning with Big Code
Professor Peter O'hearn, Facebook & UCL

Guest Speaker !

Probabilistic Programming:
What is it and how to improve it?

Maria Gorinova
University of Edinburgh

3.10pm Tuesday 29 November 2016

Optional: Review and Exam Preparation !

The final lecture will be an exam review, principally for single-semester visiting students. Everyone welcome, though. I hope to do another in Semester 2 — you arrange it, I'll be there.

Past Papers http://blog.inf.ed.ac.uk/apl16/exam#past

UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS

ADVANCES IN PROGRAMMING LANGUAGES

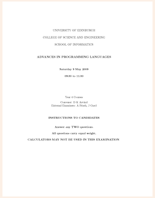
Thursday 15 May 2008
14:30 to 16:30

Year 4 Courses
Convener: D K Arvind
External Examiners: J Gurd, M Wooldridge

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.
All questions carry equal weight.
CALCULATORS MAY NOT BE USED IN THIS EXAMINATION

2007–2008


UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS

ADVANCES IN PROGRAMMING LANGUAGES

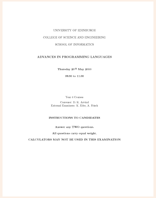
Saturday 9 May 2009
09:30 to 11:30

Year 4 Courses
Convener: D K Arvind
External Examiners: A Frisch, J Gurd

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.
All questions carry equal weight.
CALCULATORS MAY NOT BE USED IN THIS EXAMINATION

2008–2009


UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS

ADVANCES IN PROGRAMMING LANGUAGES

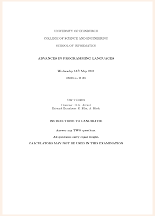
Thursday 20th May 2010
09:30 to 11:30

Year 4 Courses
Convener: D. K. Arvind
External Examiners: K. Eder, A. Frisch

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.
All questions carry equal weight.
CALCULATORS MAY NOT BE USED IN THIS EXAMINATION

2009–2010


UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS

ADVANCES IN PROGRAMMING LANGUAGES

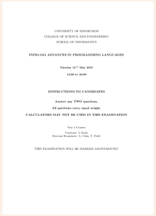
Wednesday 18th May 2011
09:30 to 11:30

Year 4 Courses
Convener: D. K. Arvind
External Examiners: K. Eder, A. Frisch

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.
All questions carry equal weight.
CALCULATORS MAY NOT BE USED IN THIS EXAMINATION

2010–2011


UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS

INFR11101 ADVANCES IN PROGRAMMING LANGUAGES

Tuesday 12th May 2015
14:30 to 16:30

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.
All questions carry equal weight.
CALCULATORS MAY NOT BE USED IN THIS EXAMINATION

Year 4 Courses
Convener: I. Stark
External Examiners: A. Cohn, T. Field

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

2014–2015

Look at the past papers. Pick out a question or lecture topic and nominate it for the review lecture mailing me or posting on Piazza.

Review

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.

Review

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.

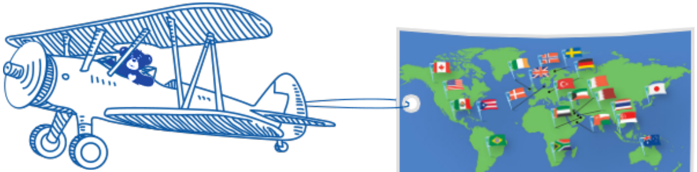
Object Deconstruction

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.

BUILD-A-BEAR WORKSHOP®



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 unitialised 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: &HasDimensions) -> i32 {
  item.height() * item.width()
}
```

Trait Objects

Here item: &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: &HasDimensions) -> i32 {
  item.height() * item.width()
}
```

Trait Objects

There is an equivalent construction on the heap with trait object Box<&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: &HasDimensions) -> i32 {
  item.height() * item.width()
}
```

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: **http://areweconcurrentyet.com/**

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, . . .

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

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?

Guest Speaker !

Probabilistic Programming:
What is it and how to improve it?

Maria Gorinova
University of Edinburgh

3.10pm Tuesday 29 November 2016

Guest Lecture: Near, Now !

Domain Specific Languages
The Xtext Framework

Avaloq Guest Lecture
Software Design and Modelling

Teviot Lecture Theatre
4.10pm Friday 25 November 2016