Introduction to Rust

Jacques-Henri Jourdan

January 25th, 2023

Low-level programming

Some software requires a high level of control:

- over memory layout;
- over when memory is allocated and freed;
- over what computations are done and when...

The standard "answer" to these issues is C/C++.
These languages have shortcomings:

- very complicated semantics (pointer optimizations...),
- no safety guarantee,
- poor support for abstraction.

Rust

A language initially developed by Mozilla for rewriting parts of Firefox

High performance, high level of control with safe abstraction mechanisms

Rust

A language initially developed by Mozilla for rewriting parts of Firefox

High performance, high level of control with safe abstraction mechanisms

Zero-cost abstraction

Powerful abstraction mechanisms with no impact on performance:

- Sound type system
- Pointers: ownership + borrows with lifetimes
- Polymorphism with traits ($\simeq$ type classes)

When the type system is not expressive enough, we can use unsafe features behind safe abstractions.

Why study Rust here?

Many concepts from type systems, adapted them to a new context:

- Polymorphism, type traits, closures, algebraic types...

New type system features:

- Ownership types, borrows, lifetimes,
- Concurrency: fine-grained tracking of thread-(un)safe types.

New challenges in type system meta-theory:

- ownership,
- unsafe code behind safe abstractions.

What you will learn here

My goal is that in the following 5 weeks, you will:

- know how to write simple Rust programs;
- roughly understand how the type system works,
  - both formally and how it can be implemented;
- have a glimpse of how to formally study Rust:
  - How to use the type system to verify Rust programs?
  - How to build a logical relation to prove soundness?

Today: learn a bit of the language

Resources

To learn Rust as a developer:

- Plenty of references here: https://www.rust-lang.org/learn
- Some of my favorites:
  - Rust 101 by Ralf Jung: accessible tutorial for the core language;
    - Many examples of this course taken from this tutorial
  - Rust by example: learn by example;
  - The Rust Book: comprehensive, but long, "official" tutorial for Rust and some of its ecosystem.

Metatheory of the type system:

- RustBelt: Securing the Foundations of the Rust Programming Language. Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer. POPL 2018.

The minimum of a sequence of integers

```
enum NumberOrNothing {
    Number(i32), Nothing
}


fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res = NumberOrNothing::Nothing;
  for el in vec {
    match res {
      NumberOrNothing::Nothing => {
        res = NumberOrNothing::Number(el);
      },
      NumberOrNothing::Number(n) => {
        let m = if n < el { n } else { el };
        res = NumberOrNothing::Number(m);
      }
    }
  }

  return res
}
```

A function for computing the maximum of a sequence of 32-bits integers.

- Parameters and return types need to be annotated.
- Types for local variables are inferred (most often).

The minimum of a sequence of integers

```
enum NumberOrNothing {
    Number(i32), Nothing
}
```

```
fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res = NumberOrNothing::Nothing;
  for el in vec {
    match res {
      NumberOrNothing::Nothing => {
        res = NumberOrNothing::Number(el);
      },
      NumberOrNothing::Number(n) => {
        let m = if n < el { n } else { el };
        res = NumberOrNothing::Number(m);
      }
    }
  }

  return res
}
```

Empty sequence
⇒ no defined minimum

We use an algebraic data type

- 32 bits integer, or nothing
- pattern matching
- constructors

The minimum of a sequence of integers

```
enum NumberOrNothing {
    Number(i32), Nothing
}
use NumberOrNothing::*;

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  let mut res =                    Nothing;
  for el in vec {
    match res {
                        Nothing => {
        res =                      Number(el);
      },
                        Number(n) => {
        let m = if n < el { n } else { el };
        res =                      Number(m);
      }
    }
  }

  return res
}
```

Empty sequence
⇒ no defined minimum

We use an algebraic data type

- 32 bits integer, or nothing
- pattern matching
- constructors
  - In a specific namespace, can be imported

The minimum of a sequence of integers

```
enum NumberOrNothing {
    Number(i32), Nothing
}
use NumberOrNoth

fn vec_min(vec:
  let mut res =
  for el in vec
    match res {

        res =
      },

        let m =
        res =
      }
    }
  }

  return res
}
```

A value of type `NumberOrNothing` is a sequence of bytes (tag followed by `i32` payload).

- Control on memory representation, like in C/C++.
- Drawback: values have sizes, which the compiler need to know.
- In higher-level like OCaml, Java, Python, Haskell, ... this is different:
  - complex values are boxed;
  - values always use one memory word;
  - unavoidable implicit indirection.

Using vec_min

```
impl NumberOrNothing {
  fn print(self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}
```

```
fn main() {
  let v = vec![18,5,7,2,9,27];
  let min = vec_min(v);
  min.print()
}
```

We can define associated functions for types:

- like sealed methods in OO languages,
- no more than a function, but without polluting the global namespace.

Records, tuples, arrays, newtype

We (of course!) have other convenient ways to build types:

- Tuples `(T1, T2, ...)`:
  - construction: `(e1, e2, ...)`
  - projections: `t.0`, `t.1`, ...
- Records declared by `struct R { f1: T1, f2: T2, ... }`
  - construction: `R { f1: e1, f2: e2, ... }`
  - projections: `r.f1`, `r.f2`, ...
- Newtype (as in Haskell) declared by `struct NT(T)`
  - construction: `NT(e)`
  - projection: `nt.0`
- Fixed-length arrays `[T; 42]`
  - construction: `[e; 42]`
  - access: `a[i]`

Records, tuples, arrays, newtype

We (of course!) have other convenient ways to build types:

- Tuples `(T1, T2, ...)`:
  - construction: `(e1, e2, ...)`
  - projections: `t.0`, `t.1`, ...
- Records declared by `struct R { f1: T1, f2: T2, ... }`
  - construction: `R { f1: e1, f2: e2, ... }`
  - projections: `r.f1`, `r.f2`, ...
- Newtype (as in Haskell) declared by `struct NT(T)`
  - construction: `NT(e)`
  - projection: `nt.0`
- Fixed-length arrays `[T; 42]`
  - construction: `[e; 42]`
  - access: `a[i]`

Again, records, tuples, enum and arrays are stored with no implicit pointer indirection

- Values of tuples, records, arrays and enums types are the concatenation of their components.

1 Basic types

2 Aliasing control
- Ownership in Rust
- Borrows and lifetimes

3 Generics & traits
- Iterators
- Closures
- Trait objects

Ownership: counter example in C++

```
void foo(std::vector<int> v) {
  int *first = &v[0];
  v.push_back(42);
  *first = 1337;
}
```

Where is the bug?

Ownership: counter example in C++

```
void foo(std::vector<int> v) {
  int *first = &v[0];
  v.push_back(42);
  *first = 1337;
}
```

- `push_back` may reallocate the vector,
  - invalidating inner pointers such as `first`.
- Somewhat perverse bug:
  - most of the time, `push_back` does not reallocate the vector.
- We mutated memory through two aliased pointers, `first` and `v`.

Bugs caused by aliasing

Iterator invalidation:

- Mutating a data structure invalidates its iterators.
  - In C++, the rule is complicated (depends on where you iterate and what mutation...).
  - Common source of bugs in C++/Java/...
  - Often undetected by the library: unexpected consequences.
  - Core of the problem: mutation and aliasing at the same time.

Double free:

- Can lead to state inconsistency of the memory management library.
- Another form of simultaneous mutation (free) and aliasing.

Data races:

- Particularly difficult to test/debug.
- By definition: mutation and aliasing.

The general aliasing rule

In Rust, we maintain the invariant: mutation XOR aliasing.

If we want to mutate, we need to have the unique alias.

- We are the owner of the memory location.
- We view memory as a resource which cannot be duplicated.

This also applies to freeing memory.

- There is no GC in Rust (better control of resources, better performances).

Ownership in vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();


}
```

Ownership in vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();
  v.push(42);
  vec_min(v).print();
}
```

Ownership in vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = ve
  vec_min(v).pri
  v.push(42);
  vec_min(v).pri
}
```

```
error[E0382]: borrow of moved value: ‘v‘
  --> src/main.rs:37:3
   |
35 |   let mut v = vec![18,5,7,2,9,27];
   |       ...
36 |   vec_min(v).print();
   |              - value moved here
37 |   v.push(42);
   |   ^^^^^^^^^^ value borrowed here after move
```

Ownership in vec_min

```
impl NumberOrNothing {
  fn print(self) { ... }
}

fn vec_min(vec: Vec<i32>) -> NumberOrNothing {
  ...
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(v).print();
  v.push(42);
  vec_min(v).print();
}
```

Passing `v` as a parameter to `vec_min` triggered a move.

- We lost its ownership. We can no longer use it.
- That's fortunate, because `vec_min` automatically freed the vector.
  - RAII: destructor is called as soon as variable leaves its scope (except if already moved).

The iconic ownership type: Box<T>

`Box<T>` is the type of owned pointers to `T`.

- Creation function:

```
fn new(x: T) -> Box<T>
```

  - Move of `x` to dynamically allocated memory (i.e., `malloc`).
  - Transfer of ownership.
- Can be dereferenced (for read/write): `*p`.
- RAII: Automatically freed when leaves the scope.

Example of use: singly linked list:

```
type List = Option<Box<Node>>;
struct Node { elem: i32, next: List, }
```

Exception to ownership tracking

Is ownership tracking a good thing for all variables?

Exception to ownership tracking

No! Some values are bare data, and can be copied freely.

- Examples: `i32`, `(i64, u64)`, `bool`, ...

We say that these types implement the `Copy` trait

- Tells the compiler to disable ownership tracking for values of these types.

By default, types declared with `struct` or `enum` are not `Copy`.

- Because they could be used to represent resources not know by the compiler.
- To make it so, this is easy:

```
#[derive(Copy, Clone)]
enum NumberOrNothing {
    Number(i32), Nothing
}
```

- (More explanations on traits later.)

Shared borrows

Variant of `vec_min` to avoid the move:

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

Instead of passing the ownership of the vector we pass a shared borrow of it.

- At runtime: a pointer to `v`
- Here, the shared borrow is only valid for the duration of the function.
- `&T` implements `Copy`.
- `&T` does not allow mutation (usually... see next course).
- aka. immutable borrow, immutable reference.

Shared borrows

Variant of `vec_min` to avoid the move:

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

What is the type of `el`? Why?

Shared borrows

Variant of `vec_min` to avoid the move:

```
fn vec_min(vec: &Vec<i32>) -> NumberOrNothing {
  let mut res = Nothing;
  for el in vec {
    match res {
      Nothing => { res = Number(*el); },
      Number(n) => {
        let m = if n < *el { n } else { *el };
        res = Number(m);
      }
    }
  }
  return res
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_min(&v).print();
  v.push(42);
  vec_min(&v).print();
}
```

When we access elements of a shared borrow of a vector (e.g., by iterating with a `for` loop), we get shared borrows of the content, because, in general, we cannot transfer ownership out of the vector.

Thus we need to dereference `el`!

Unique borrows

```
fn vec_inc(vec: &mut Vec<i32>) {
  for el in vec {
    *el += 1
  }
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_inc(&mut v);
  v.push(42); /* ... */
}
```

Unique borrows are similar to shared borrows.

Differences:

- Allows mutation.
- ...

Unique borrows

```
fn vec_inc(vec: &mut Vec<i32>) {
  for el in vec {
    *el += 1
  }
}

fn main() {
  let mut v = vec![18,5,7,2,9,27];
  vec_inc(&mut v);
  v.push(42); /* ... */
}
```

Unique borrows are similar to shared borrows.

Differences:

- Allows mutation.
- Does not implement `Copy`.
- aka. mutable borrow, mutable reference.

Borrows and methods

```
impl NumberOrNothing {
  fn print( self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}
```

Call:

- copy of `self`
- full ownership transfer

Can we pass a borrow for `self`?

Borrows and methods

```
impl NumberOrNothing {
  fn print(&self) {
    match self {
      Nothing =>
        println!("The number is: <nothing>"),
      Number(n) =>
        println!("The number is: {}", n),
    };
  }
}
```

We can tell that `print` takes a borrow to `self`.

Same syntax at call site:

```
let n = Nothing;
n.print();
```

Borrowing is automatic at call-site for `self`.

Of course, this also works with `&mut`.

Pointers in data structures

Imagine we have a vector of non-`Copy` elements. Say, `Vec<Vec<i32>>`.
How do we mutate the content?

One possibility:

```
fn set(i: i32, v: Vec<i32>, l: &mut Vec<Vec<i32>>)
```

The problem: cannot modify a `Vec<i32>` which is already in the vector!
We would need to move out the `Vec<i32>` and then move it in back. Not very efficient.

Pointers in data structures

Imagine we have a vector of non-`Copy` elements. Say, `Vec<Vec<i32>>`.
How do we mutate the content?

Another possibility

```
fn get_mut(i: i32, l: &mut Vec<Vec<i32>>) -> &mut Vec<i32>
```

but we said that borrows end when function terminate...

- We cannot use the return value

Lifetimes

Borrows have lifetimes, noted `'a`, `'b`, `'c`...

- In general, for a borrow we write `&'a T` or `&mut 'a T`.
  - In some situations, this can be abbreviated to `&T` / `&mut T`.

Functions can be lifetime polymorphic to return inner pointer.
Example adapted from `Vec` library:

```
fn last_mut<'a, T>(v: &'a mut Vec<T>) -> Option<&'a mut T>
```

- Returns a borrow with the same lifetime than the parameter
- The calling function can update the content of the vector, as long as it does not access the vector at the same time
  - This preserves the ownership invariant (mutation XOR aliasing)
- The lifetime is automatically inferred when calling the function

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);



  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }


}
```

- When creating a borrow (with `&mut v`), Rust creates a lifetime variable `'a`.
- `v` is marked as mutably borrowed for `'a`.
  - `v` cannot be used as long as `'a` is alive.
- This lifetime variable goes through type inference.
  $\Longrightarrow$ variable `last` has type `&'a mut i32`
- We have only one constraint for `'a`: be alive when `last` is used
  $\Longrightarrow$ `'a` is inferred to be the zone.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);



  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }

  v.push(4)
}
```

Let's add a mutation of `v` after the end of `'a`.

Rust checks the status of `v` at the call.
$\Longrightarrow$ The lifetime has ended, `v` is available.
$\Longrightarrow$ Call allowed.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec<i32> = vec![1,2,2];

  let opt = last_mut(&mut v);

  v.push(42); // Danger!

  match opt {
    Some(last) => *last = 3,
    None => panic!()
  }

}
```

If we try to mutate v when it is still borrowed

Rust checks the status of v at the call.
The lifetime has NOT ended, the call is impossible.

Lifetimes, example

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

fn f() {
  let mut v: Vec

  let opt = last

  v.push(42); //

  match opt {
    Some(last) =
    None => pani
  }


}
```

If we try to mutate v when it is still borrowed

```
error[E0499]: cannot borrow ‘v‘ as mutable more than once at a time
  --> src/lib.rs:10:3
   |
8  |   let opt = last_mut(&mut v);
   |                      ------ first mutable borrow occurs here
9  |
10 |   v.push(42); // Danger!
   |   ^^^^^^^^^^ second mutable borrow occurs here
11 |
12 |   match opt {
   |         --- first borrow later used here
```

Basic operations on borrows

Copying a shared borrow (recall: `&T: Copy`):

```
let x = &1;
let y = x;
println!("{} {}", *x, *y);
```

Downcasting mutability:

```
let x = &mut 1;     // x: &'a mut i32
let y: &i32 = x;  // y: &'a i32
```

Splitting a borrow (shared or unique):

```
let x = &mut (1, 2);  // x: &'a mut (i32, i32)
let (x1, x2) = x;     // x1: &'a mut i32   x2: &'b mut i32
```

Directly on a pair/record/enum:

```
let mut p = (42, 12);
let x = &mut p.0;
let y = &p.1; // Allowed even if p.0 is uniquely borrowed
let z = &p.1; // Allowed even if p.1 is already borrowed immutably
let t = p.1;  // Idem
*x = 43; // x: &mut i32
println!("{} {} {}", *x, *y, *z); // y, z: &i32
```

Reborrowing

```
fn last_mut<'a, T>(v: &'a mut Vec<T>)
  -> Option<&'a mut T>
{ ... }

let v = &mut vec![1, 2, 3];
match last_mut(&mut(*v)) { ... }
match last_mut(&mut(*v)) { ... }
```

Reborrowing creates a shorter borrow of the content of a borrow.
The old borrow is reactivated when the new borrow ends.

- Allows using a mutable borrow several times in a row.
- Happens implicitly in the vast majority of cases.
- The lifetime of the original borrow needs to be longer than the reborrowing lifetime.

Lifetime subtyping

Reborrowing uses a notion of lifetime inclusion

- `'a` outlives `'b`, written `'a: 'b`.

This order relation: lifetime subtyping.

- The only source of subtyping in Rust.
- Constrained lifetime polymorphism:

```
fn f<'a, 'b: 'a>(...) -> .. { ... }
fn f<'a, 'b>(...) -> .. where 'b: 'a { ... }
```

- Exercise: what are the variances of `&`? `&mut`? `Box<` `>`? `Vec<` `>`?
  - With respect to the lifetime parameter?
  - With respect to the type parameter?

Lifetime inference

Most of the lifetime information is inferred by the compiler.

- We only need to annotate function types (and type declarations).

Key component of `rustc`: the borrow checker.

- Runs after traditional type-checking, and uses information from it.
- Uses variable liveness information from dataflow analysis.
- Tracks ownership (e.g., `Box<T>` value used only once) and infers lifetimes.
- Implemented on MIR, an intermediate representation in CFG form.

The borrow checker

Lifetime inference

The borrow checker interprets each lifetime `'a` as a set $[\![$`'a`$]\!]$ containing:

- nodes of the CFG,
- elements `end('a)` for lifetimes `'a` generalized at the function's prototype.

Type-checking generates fresh lifetime variables and outlives constraints "`'a: 'b`":

- Examples: reborrowing, function calls, coercions, ...
- Interpreted as set inclusion $[\![$`'b`$]\!] \subseteq [\![$`'a`$]\!]$

Further constraints are added:

- `L` $\in [\![$`'a`$]\!]$ when `'a` appears in the type of a live variable at CFG node `L`;
- `end('a)` $\in [\![$`'a`$]\!]$ for any universal variable `'a`;
- `L` $\in [\![$`'a`$]\!]$ for any universal variable `'a` and any CFG node `L`.

Borrow checker finds the smallest solution of the set of constraints (fixpoint algorithm).

The borrow checker

Checking the program

Once sets $[\![$`'a`$]\!]$ are computed, it remains to check:

- that variables accesses are valid:
  - No read if not initialized or moved out.
  - No access if existing mutable loan with alive lifetime.
  - No write if existing immutable load with alive lifetime.
  - Question: when should borrowing be allowed?

- that outlives constraints at the function prototype are sufficient:
  - For any universally quantified lifetimes `'a` and `'b`, if `end('a)` $\in [\![$`'b`$]\!]$, then outlives constraints should imply `'b: 'a`.

The borrow checker

Checking the program

Once sets $[\![\texttt{'a}]\!]$ are computed, it remains to check:

- that variab
  - No rea
  - No acc
  - No wri
  - Questi
- that outlive
  - For any universally quantified lifetimes `'a` and `'b`, if $\texttt{end('a)} \in [\![\texttt{'b}]\!]$, then outlives constraints should imply `'b: 'a`.

Additional complications in `rustc`:

- Instead of variables: places
  - E.g., `rustc` treats `p.0` and `p.1` independently.
- Language features: closures...
- Efficient algorithm for fixpoint computation.
- ...

Generics (aka. polymorphism)

What if we want to have a generic version of `vec_min`?

```
enum Option<T> { // Already in stdlib
  None,
  Some(T),
}
use Option::*;
```

Generics (aka. polymorphism)

What if we want to have a generic version of `vec_min`?

```
enum Option<T> { // Already in stdlib
  None,
  Some(T),
}
use Option::*;

fn vec_min<T>(vec: Vec<T>) -> Option<T> {
  let mut res = None;
  for el in vec {
    match res {
      None => { res = Some(el); },
      Some(n) => {
        let m = if n < el { n } else { el };
        res = Some(m);
      }
    }
  }

  return res
}
```

Generics (aka. polymorphism)

What if we want to have a generic version of `vec_min`?

```
enum Option<T> { // Already in stdlib
  None,
  Some(T),
}
use Option::*;

fn vec_min<T>(ve
  let mut res =
  for el in vec
    match res {
      None => {
      Some(n) =>
        let m =
        res = So
      }
    }
  }

  return res
}
```

Does this code compiles?
Is there something missing?

Generics (aka. polymorphism)

What if we want to have a generic version of `vec_min`?

```
enum Option<T> { // Already in stdlib
  None,
  Some(T),
}
use Option::*;

fn vec_min<T>(ve
  let mut res =
  for el in vec
    match res {
      None => {
      Some(n) =>
        let m =
        res = So
      }
    }
  }

  return res
}
```

```
error[E0369]: binary operation `<` cannot be applied to type `T`
  --> src/lib.rs:13:22
   |
13 |         let m = if n < el { n } else { el };
   |                   - ^ -- T
   |                   |
   |                   T
   |
help: consider restricting type parameter `T`
   |
7  | fn vec_min<T: std::cmp::PartialOrd>(vec: Vec<T>) -> Option<T> {
   |             ++++++++++++++++++++++
```

Generics (aka. polymorphism)

We have generics with trait bounds:

```
enum Option<T> { // Already in stdlib
  None,
  Some(T),
}
use Option::*;

fn vec_min<T: std::cmp::PartialOrd>(vec: Vec<T>) -> Option<T> {
  let mut res = None;
  for el in vec {
    match res {
      None => { res = Some(el); },
      Some(n) => {
        let m = if n < el { n } else { el };
        res = Some(m);
      }
    }
  }

  return res
}
```

Traits (Rust's type classes)

Rust's type traits resemble Haskell's type classes:

- A mechanism for overloading:
  - give different meaning to the same function name or symbol, depending on parameters and return types.
- A mechanism for abstraction:
  - the same function works for different implementations of some interface.

Traits (Rust's type classes)

Rust's type traits resemble Haskell's type classes:

- A mechanism for overloading:
  - give different meaning to the same function name or symbol, depending on parameters and return types.
- A mechanism for abstraction:
  - the same function works for different implementations of some interface.

`Self`: special type parameter for the `self` parameter of methods.

We can declare traits/type classes:

```
trait Trait<X1, X2, ...>: Trait0<...> {
  type AssocTy;
  fn method1(self, ...) -> Ty1;
  fn method2(&self, ...) -> Ty2 {
    // Default implementation
  }
  fn method3(x: i32, ...) -> Self;
  ...
}
```

```
class (C0 self ...) => C self a b ... where
  data AssocTy self a b ... :: *;
  method1 :: self -> ... -> ty1
  method2 :: self -> ... -> ty2
  method2 s ... =
    -- Default implementation
  method3 :: Int32 -> ... -> self
  ...
```

Traits (Rust's type classes)

Rust's type traits resemble Haskell's type classes:

- A mechanism for overloading:
  - give different meaning to the same function name or symbol, depending on parameters and return types.
- A mechanism for abstraction:
  - the same function works for different implementations of some interface.

`Self`: special type parameter for the `self` parameter of methods.

We can write instances of them:

```
impl<X1, ...> Trait<Ty1, ....> for Ty
    where Ty2: Trait1, ... {

  type AssocTy = ...;
  fn method1(self, ...) -> Ty1 {
     ...
  }
  ...
}
```

```
instance (C1 ty2) => C ty1 ... where


  data AssocTy self ty1 ... = ...
  method1 s ... = ...

  ...
```

Traits (Rust's type classes)

Rust's type traits resemble Haskell's type classes:

- A mechanism for overloading:
  - give different meaning to the same function name or symbol, depending on parameters and return types.
- A mechanism for abstraction:
  - the same function works for different implementations of some interface.

`Self`: special type parameter for the `self` parameter of methods.

We can use them:

```
fn f<X>(...) -> ... where T: Trait {
// Same as: fn f<X: Trait>(...) -> ...
  ...
  x.method1(a, b, c)
  ...
}
```

```
f :: C x => ...

f ... = ... method1 x a b c ...
```

Function vec_min requires T: PartialOrd

```
fn vec_min<T: PartialOrd>(vec: Vec<T>) -> Option<T> {
  ...
      if n < el { n } else { el };
  ...
}

trait PartialOrd { // Simplified
  fn partial_cmp(&self, other: &Self) -> Option<Ordering>;
  fn lt(&self, other: &Self) -> bool { ... }
  ...
}
```

Function vec_min requires T: PartialOrd

```
fn vec_min<T: PartialOrd>(vec: Vec<T>) -> Option<T> {
  ...
      if n < el { n } else { el };
  ...
}

trait PartialOrd { // Simplified
  fn partial_cmp(&self, other: &Self) -> Option<Ordering>;
  fn lt(&self, other: &Self) -> bool { ... }
  ...
}
```

Function `vec_min` is specialized for every type `T` with which it is used.
Hence, the call to `lt` is statically resolved (and can be inlined if necessary).

Comparison with Haskell's type classes

`Self` parameter

One of the trait parameter is implicit: `Self`.

- Just syntax, no real expressiveness change

Monomorphisation

Traits do not use indirect calls via dictionnaries like in Haskell.
Instead, generic functions are specialized for each use, and trait method calls are statically resolved.
This is a form of zero cost abstraction.

Higher-kinded parameters

Example: in `Monad M`, we have `M :: * -> *`.
In Rust, only types are allowed in trait parameters.

Coherence, orphan rules

There cannot be two conflicting instances in the same program.
This is important for soundness of some libraries.

Problem: how to enforce coherence when there are several compilation units?

Answer: "orphan rules".

- Guarantees: no conflicting instances exist in two different compilation units.
- Roughly: one can declare an instance if trait or the implemented type is declared in the current trait.
- Details are complicated by generic types, parameterized traits, ...

Traits as a way of asserting properties

Some traits are used for expressing properties of types.
They do not have any method.

Examples:

- `Copy` expresses that a type can be copied without ownership issues.
- `Sized` expresses that the compiler knows the size of a type.
- `Send` and `Sync` express that a type is thread-safe (in some sense).

These traits have restrictions to which type can implement them:

- some of them are `unsafe`,
- others are treated specially by the compiler,
- some of them are automatically implemented (but we can opt-out).

An example of use of traits: `Iterator` and `IntoIterator`

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
  ...
}

trait IntoIterator {
  type Item;
  type IntoIter: Iterator<Item = Self::Item>;
  fn into_iter(self) -> Self::IntoIter;
}
```

Syntactic sugar:

```
for x in c {
  ...
}
```

$\Longrightarrow$

```
let mut it = c.into_iter();
loop {
  match it.next() {
    None => break,
    Some(x) => {
      ...
    }
  }
}
```

An example of use of traits: Iterator and IntoIterator

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
  ...
}

trait IntoI
  type Item
  type Into
  fn into_i
}
```

Syntactic s

```
for x in c {
  ...
}
```

$\Longrightarrow$

```
match it.next() {
  None => break,
  Some(x) => {
    ...
  }
}
}
```

Important to note: `Iterator` is not a type!

There is a different type for each implementation of `Iterator`.
This type describes the specialized state of the iterator. It is different depending on the data structure we are iterating over.

Very different to e.g., the type `Seq.t` in OCaml.

Ranges

- Ranges such as `0..10` implement `IntoIterator` and `Iterator`.
- `for i in 0..N {...}` is easily optimized.
  - This is actually the usual way of writing `for` loops.

Ranges

- Ranges such as `0..10` implement `IntoIterator` and `Iterator`.
- `for i in 0..N {...}` is easily optimized.
  - This is actually the usual way of writing `for` loops.

```
// The type behind the 0..N syntax
// (Specialized to i32 for simplicity)
struct Range {
  start: i32,
  end: i32,
}

impl Iterator for Range {
  type Item = i32;

  fn next(&mut self) -> Option<i32> {
    if self.start < self.end {
      let r = self.start; self.start += 1;
      return Some(r)
    } else {
      return None
    }
  }
}
```

Ranges

- Ranges such as `0..10` implement `IntoIterator` and `Iterator`.
- `for i in 0..N {...}` is easily optimized.
  - This is actually the usual way of writing `for` loops.

The code written by the user:

```
for i in 0..N {
  ...
}
```

Ranges

- Ranges such as `0..10` implement `IntoIterator` and `Iterator`.
- `for i in 0..N {...}` is easily optimized.
  - This is actually the usual way of writing `for` loops.

Removing syntactic sugar:

```
let r = Range{ start: 0, end: N };
let mut it = r; // IntoIterator is trivial for Range
loop {
  match it.next() {
    None => break,
    Some(i) => {
      ...
    }
  }
}
```

Ranges

- Ranges such as `0..10` implement `IntoIterator` and `Iterator`.
- `for i in 0..N {...}` is easily optimized.
  - This is actually the usual way of writing `for` loops.

After inlining `next`:

```
let r = Range{ start: 0, end: N };
let mut it = r;
loop {
  let o =
    if it.start < end {
      let r = it.start; it.start += 1;
      Some(r)
    } else { None };
  match o {
    None => break,
    Some(i) => {
      ...
    }
  }
}
```

Ranges

- Ranges such as `0..10` implement `IntoIterator` and `Iterator`.
- `for i in 0..N {...}` is easily optimized.
  - This is actually the usual way of writing `for` loops.

After inverting `if` and `match`, and simplifications:

```
let mut it = Range{ start: 0, end: N };
loop {
  if it.start < it.end {
    let i = it.start;
    it.start += 1;
    ...
  } else {
    break
  }
}
```

Very close to what one would expect from a C/C++ `for` loop!
It remains to:

- move `it.start += 1` to the end of the loop;
- use appropriate memory (stack or registers) for `it.start` and `it.end`.

Iterating over Vec<T>

There exists three ways to iterate over a vector:

```
for x in v { ... }      // x:T,      full ownership,     consumes v.
for x in &mut v { ... } // x:&mut T, unique borrow of v, changes elements of v.
for x in &v { ... }     // x:&T,     shared borrow of v, leaves v as-is.
```

Three different instances for `IntoIterator` (simplified):

```
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> { ... }
}

impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    type IntoIter = Iter<'a, T>;
    fn into_iter(self) -> Iter<'a, T> { ... }
}

impl<'a, T> IntoIterator for &'a mut Vec<T> {
    type Item = &'a mut T;
    type IntoIter = IterMut<'a, T>;
    fn into_iter(self) -> IterMut<'a, T> { ... }
}
```

Iterating over Vec<T>

There exists three ways to iterate over a vector:

```
for x in v { ... }      // x:T,      full ownership,     consumes v.
for x in &mut v { ... } // x:&mut T, unique borrow of v, changes elements of v.
for x in &v { ... }     // x:&T,     shared borrow of v, leaves v as-is.
```

```
impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    type IntoIter = Iter<'a, T>;
    fn into_iter(self) -> Iter<'a, T> { ... }
}
```

In borrow mode, the iterator (e.g., `Iter<'a, T>`) contains a borrow to the vector!

$\Longrightarrow$ It depends on a lifetime `'a`.

$\Longrightarrow$ `v` cannot be used as long as the iterator is alive.

$\Longrightarrow$ No iterator invalidation!

Closures

Traditionally, in functional languages, closures are the runtime representation of values of the function type.

In Rust, two adjustments:

- The notion of function type is too dynamic.
  Unsized $\Longrightarrow$ manipulated through a pointer.
  Contains function pointer $\Longrightarrow$ indirect function calls (slow, inlining difficult).
  - In Rust, functions are a trait, not a type.
- Need to take into account ownership of captured variables.
  - Three traits for closures: `Fn`, `FnMut` and `FnOnce`.

Closures

The FnOnce trait

```
trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}
```

When the user writes `|a: &mut u64, b: i32| ...`, Rust generates during typing:

- an anonymous `struct` type, with one field for each captured variable,
- an instance of `FnOnce<(&mut u64, i32)>` for this anonymous `struct` type
  - `Output`: the return type of the function,
  - `call_once`: the body of the function (captured variables: via `self`),
- code which creates and initializes the `struct` of captured variables.

Notes:

- The `call_once` method gets full ownership of `self`.
  - The body of the closure can get full ownership of captured variables.
  - The closure can be called only once!
- `Args` is in fact a tuple of parameters.
- Notation for this trait: `FnOnce(&mut u64, i32) -> Option<u64>`

Closures

The FnOnce trait

```
trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}
```

When the user writes `|a: &mut u64, b: i32| ...`, Rust generates during typing:

- an anonym
- an instance
  - Output
  - call_
- code which

This is a form of closure conversion, but during typing.

Each closure gets a different type, but they all implement `FnOnce`.

Notes:

- The `call_once` method gets full ownership of `self`.
  - The body of the closure can get full ownership of captured variables.
  - The closure can be called only once!
- `Args` is in fact a tuple of parameters.
- Notation for this trait: `FnOnce(&mut u64, i32) -> Option<u64>`

Closures

The FnOnce trait, example

`std::iter::once(x)`: an iterator which returns `x`, only once.

```
fn once<A>(value: A) -> Once<A>
impl<A> Iterator for Once<A> { type Item = A; ... }
```

Closures

The FnOnce trait, example

`std::iter::once(x)`: an iterator which returns `x`, only once.

```
fn once<A>(value: A) -> Once<A>
impl<A> Iterator for Once<A> { type Item = A; ... }
```

A variant: `std::iter::once_with(f)`.
The same, but lazily, through a closure.

```
fn once_with<A, F>(gen: F) -> OnceWith<F>
  where F: FnOnce() -> A
impl<A, F: FnOnce() -> A> Iterator for OnceWith<F> {
  type Item = A;
  ...
}
```

Note: generic with respect to the closure type `F`!

Call example:

```
let it = std::iter::once_with(|| 42);
println!("{}", it.next());
```

Closures

Three traits: `FnOnce`, `FnMut`, `Fn`

```
trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

trait FnMut<Args>: FnOnce<Args> { // Sub-trait
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

trait Fn<Args>: FnMut<Args> { // Sub-trait
    fn call(&self, args: Args) -> Self::Output;
}
```

- Rust automatically generates as many instances as possible when using the `|...|` `...` syntax.
- Captured variables are moved/mutably borrowed only if necessary.

Examples of APIs with closures

Iterator combinators

There exists many combinators for iterators.
Examples:

```
trait Iterator {
...
  fn map<B, F>(self, f: F) -> Map<Self, F>
    where F: FnMut(Self::Item) -> B { ... }
  fn filter<P>(self, predicate: P) -> Filter<Self, P>
    where P: FnMut(&Self::Item) -> bool { ... }
...
}

impl<B, I: Iterator, F> Iterator for Map<I, F>
where F: FnMut(I::Item) -> B,
{ type Item = B; ... }

impl<I: Iterator, P> Iterator for Filter<I, P>
where P: FnMut(&I::Item) -> bool,
{ type Item = I::Item; ... }
```

We can write, for example:

```
(0..N).filter(|i| *i % 2 == 0).map(|i| i * i).sum()
```

Zero-cost abstraction with iterators and closures

Let's see how the following piece of code gets compiled:

```
(0..N).filter(|i| *i % 2 == 0).map(|i| i * i + a).sum()
```

We first expand closures...

Zero-cost abstraction with iterators and closures

After expanding closures:

```
struct C1 { /* No captured variable */ }
impl FnOnce<(&i32,)> for C1 { type Output = i32; ... }
impl FnMut<(&i32,)> for C1 {
  fn call_mut(&mut self, args:(&i32,)) -> bool { *args.0 % 2 == 0 }
}

struct C2 { a: i32 }
impl FnOnce<(i32,)> for C2 { type Output = i32; ... }
impl FnMut<(i32,)> for C2 {
  fn call_mut(&mut self, args:(i32,)) -> i32 { args.0 * args.0 + self.a }
}

(0..N).filter(C1 {}).map(C2 { a: a }).sum()
```

We now inline `filter`, `map` and `sum`...

Zero-cost abstraction with iterators and closures

After inlinling `filter`, `map` and `sum`:

```
/* Definition and instances of C1 and C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter:
    Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  match it.next() {
    None => break,
    Some(i) => r += i
  }
}
```

We now inline <Map>::next...

Zero-cost abstraction with iterators and closures

After inlining <Map>::next:

```
/* Definition and instances of C1 and C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter:
    Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  let o = match it.iter.next() { None => None, Some(i) => Some(it.f.call_mut(i)) };
  match o {
    None => break,
    Some(i) => r += i
  }
}
```

Important: `call_mut` is a direct call!

We can merge the two `match` constructs, and inline <C2>::call_mut...

Zero-cost abstraction with iterators and closures

After inlining <C2>::call_mut and simplifications:

```
/* Definition and instances of C1 and C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter:
    Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  match it.iter.next() {
    None => break,
    Some(i) => r += i*i+it.f.a
  }
}
```

Similarly, we can inline <Filter>::next and <C1>::call_mut...

Zero-cost abstraction with iterators and closures

After inlining <Filter>::next and <C1>::call_mut:

```
/* Definition and instances of C1 and C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter:
    Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  let o;
  loop { match it.iter.iter.next() {
    None => { o = None; break },
    Some(i) => if i % 2 == 0 { o = Some(i); break }
  } }
  match o {
    None => break,
    Some(i) => r += i*i+it.f.a
  }
}
```

In the CFG graph, it is easy to see that the two `match` constructs can be merged...

Zero-cost abstraction with iterators and closures

We merged the two `match` constructs, and simplified:

```
/* Definition and instances of C1 and C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter:
    Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  match it.iter.iter.next() {
    None => break,
    Some(i) => if i % 2 == 0 { r += i*i+it.f.a }
  }
}
```

We inline `<Range>::next()` and expands the parts of `it` that stay unchanged...

Zero-cost abstraction with iterators and closures

... and we finally obtain:

```
let mut range_start = 0;
loop {
  if range_start < N {
    let i = range_start;
    range_start += 1;
    if i % 2 == 0 { r += i*i+a }
  } else { break }
}
```

We started with this concise and "abstract" piece of code:

```
(0..N).filter(|i| *i % 2 == 0).map(|i| i * i + a).sum()
```

And ended-up with a well-optimized version!
No optimization step was difficult, any optimizing compiler performs these!

An example of zero-cost abstraction!

Trait objects

Sometimes, a true function type is really needed.

- Because too much specialization increases generated code size.
- Because sometimes the set of required specialization is unbounded (ex. code generated after CPS transform).

Solution: some traits can be turned into a type: trait object.

- Can be seen as a bounded existential type.
- Syntax: `dyn Trait`
- Trait objects types are unsized.
  - Size cannot be determined statically, depends on the implementing type.
  - Only used with (fat) pointers: `Box<dyn Trait>`, `&dyn Trait`, ...
- Method calls are performed indirectly, through a vtable.
  - Some methods are incompatible: generics, using `Self` without a pointer...
    $\Longrightarrow$ only object safe traits are allowed in a trait object.

Trait objects

What type would you use for continuations in CPS?

Trait objects

What type would you use for continuations in CPS?

We could use `Box<dyn FnOnce(...) -> ...>`.

Conclusion: Rust uses the same abstraction mechanism for both static dispatch (i.e., templates in C++) and dynamic dispatch (i.e., virtual methods in C++).