Rust: Generics and Traits

Jacques-Henri Jourdan

February 4th, 2025

Project

Do not forget your project.

Deadline: Friday, February, 28th.

Send it to the following email address:
jacques-henri.jourdan@universite-paris-saclay.fr
(My email address `@cnrs.fr` has bad "security" filters which removes your attachment.)

Do not hesitate to ask questions early.

Abstract from last time (10/12/2024)

Last time, we saw many features of the Rust programming language:

- algebraic data types, records;
- various kinds of pointers: `Box<T>`, `&mut T`, `&T`;
- how Rust controls ownership and aliasing, the concept of lifetime;
- Unsafe blocks/functions: workaround strong static type-checking constraints,
  - but correct unsafe code: very subtle.
- Encapsulation: clients can safely use libraries written with unsafe code.
- Interior mutability (the ability to mutate through shared borrows): a typical example of well-encapsulated unsafe code.

Example from last week

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

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

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 X: 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

```rust
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++).