Zero-Cost Abstractions

Luca Abeni
luca.abeni@santannapisa.it

March 13, 2020

Types and Operations

- Rust custom types are new types defined by the user
- A data type defined by:
  - Set of possible values the variable can contain
  - Operations such values
- Until now, we only saw how to define the possible values (`struct` and `enum`)
- Let's define the operations for a custom type!
  - `impl` block: methods and associated functions
  - Applied to `struct`, recalls concepts from OO design...
  - Applied to `enum`, is a less known concept

Methods

- Method: function associated to a custom type (`struct` or `enum`)
- Always bound to a variable (of the method's custom type)
    - First parameter of the method: `self` value (or reference to it)
    - It is the variable (of the custom type) used to invoke the method
    - Note: it can be a value, a reference, or a mutable reference!
- If the custom type is a struct, looks like a class method in C++, Java or similar
- But impl blocks can be used for enum types too...

Methods — Example

```
struct Point {
  x: f64,
  y: f64
}

impl Point {
  fn display(&self) {
    println!("({},{})", self.x, self.y)
  }
}

fn main()
{
  let p = Point{x:1.0,y:1.0};
  p.display();
}
```

Methods on Enumerations

```
enum Colore {
    Bianco,
    Nero
}

impl Colore {
    fn stampa(&self) {
        match self {
            Colore::Bianco => println!("Bianco!"),
            Colore::Nero   => println!("Nero!")
        }
    }
}

fn main() {
    let v = Colore::Nero;

    v.stampa()
}
```

More on impl Blocks

- Multiple `impl` blocks can be added to the same data type
- An `impl` block can contain definition of functions that are not methods
  - Associated functions: do not have `self`
  - Associated to the type, but not bound to any particular variable
- Example: “`new`” function
  - It is not a method or a constructor

```
impl Point {
  fn new(v:f64) -> Point{
    Point{x:v, y:v}
  }
}
```

Abstractions and Overhead

- Zero cost abstractions (from C++): what you do not use, you do not pay for
  - Introduce only the overhead of the abstractions that are really used
  - How?
- Rust approach (once again): resolve as much as possible at build time!
  - Avoid dynamic method dispatch
  - Avoid duck typing
  - ...
- In general, specify the types behaviour so that the compiler knows it!

Specifying the Types Behaviour

- Mechanism used by Rust: traits
  - Mechanism used to define shared behaviours in an abstract way
  - Similar to interfaces, but with some important differences
- Behaviour of a type: set of methods invocable on the type, + some other properties...
  - For example, the fact that for this type assignments have a copy behaviour!!!
- Trait syntax: "`trait`" keyword followed by a "{}" block
  - Can contain declarations (an maybe definitions...) or methods and associated functions

Differences between Traits and Interfaces

- An interface is generally specified for new data types when defining them
  - The interface of pre-defined types cannot be modified/extended
- Traits can be implemented for existing types after they are defined
  - First I define a new structure "`struct S`"...
  - ...Then I define a trait "`trait Display`"...
  - ...And finally I implement "`Display`" for "`S`"!
- Traits can be implemented even for pre-defined types...
  - I can implement "`Display`" even for "`i32`"!

Trait Example

```
struct S {
  v1: f64, v2: f64
}
trait Display {
  fn display(&self);
}
impl Display for S {
  fn display(&self) {
    println!("This␣is␣an␣S({},{})", self.v1, self.v2)
  }
}
impl Display for i64 {
  fn display(&self) {
    println!("This␣is␣a␣64␣bit␣integer:␣{}", self);
  }
}

fn main()
{
  let s = S{v1:1.1,v2:1.1};
  let n1 = 2;

  s.display();
  n1.display()
}
```

More on Traits

- A trait generally declares some methods/functions to be implemented for the type...
- ...But in some cases it can also define the methods/functions!
  - Provide default implementation...
- The default implementation is used when "`impl ...`" is used for a type without specifying the method implementations (empty "`impl`" block, etc...)
- In some cases, empty traits also make sense
  - We will see later...

Using Traits

- Interesting concept, but... What are traits useful for?
  - Defining/extending the interface of a type
  - Declaring the properties of a type
  - All done at build time!
- The real power of traits becomes clear only when considering generic functions and types
  - Will see later
- Can be used to define functions that accept different types as input
  - Defining some properties of the input types
  - Example: "`fn f(v: impl Display) { ...`"

Trait Parameters...

```
fn f(v: impl Display) {
  println!("Going␣to␣invoke␣display():");
  v.display()
}
```

- For the parameter "`v`", no concrete type is specified
- "`impl Display`" here denotes a generic type for which the "`Display`" trait is implemented
  - Hence, "`f()`" can invoke "`v.display()`"
- The compiler knows how "`f()`" is invoked...
  - ...And can generate different versions of the function (one for "`S`", one for "`i64`", ...)
  - There is no runtime cost/overhead!

...And Generic Functions!

- In the previous example, "`f()`" is a *generic function*
  - Can receive multiple types for the input parameter
  - One single function definition, using the trait interface of a generic type
- Monomorphized by the compiler at build time
  - A different version of the function is generated, for each type used to invoke it
- Similar to C++ templates
- This is just a special case of generic function (generic types exist too!!!)

Generic Functions and Types

- Functions like "`fn f(v: impl Display)`" are generic
  - The code describes *a classes* of functions, all with the same structure but operating on different concrete types
- Here, the generic nature of the function is hidden...
- ...But it can be made more explicit
- "`fn f<T: Display>(v: T)`"
  - See? The function is parametric respect to type "`T`"!
  - Similar to C++ templates...
- Parametric types (generic types) exist too...

Generics Syntax

- Inspired by the C++ templates syntax:
  - The “type parameter” is part of the function/type name
  - Enclosed in angle brackets
- So, “`id<T>(v: T) -> T`” is a generic function with type parameter “`T`”
- Multiple (comma-separate) type parameters are of course possible

```
struct S<T,V> {
  v1: T,
  v2: V
}
```

Remember the Option Type?

- Sum type, previously introduced to avoid NULL pointers...
  - Values "`Nothing`" or "`Just(p)`"
- Can be more generic, not only for pointers
- Here is a possible definition in Rust:

```
enum Option<T> {
    Some(T),
    None
}
```

- Why "`None`" and "`Some()`" instead of "`Nothing`" and "`Just()`"?
  - Because these are the names actually used by Rust

Predefined Generic Types

- Rust provides some useful generic types like "`Option<T>`"
  - Not really predefined, they are part of a Rust standard library
- Other important type (another generic sum type): "`Result`"

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

- Used by all the standard functions that can potentially fail
- Two type parameters: "`T`" (the wrapped result) and "`E`" (describing a possible error)

Monomorphization

- As usual, Rust tries to minimize the abstractions' overhead
- Done through *monomorphization* at build time (as for traits)
  - The generic code/type is transformed in specific instances of the function/type by replacing type parameters with concrete types
  - Again, the compiler knows the concrete values of type parameters when the generic function is called (or the generic type is used)
- Example: "`f<T>(v: T) -> T`" invoked as "`f(3)`" and "`f(3.14)`"...
  - The compiler generates 2 functions: one working on integers and one on floating points

Scope and Lifetime

- Scope of an “entity” (value or variable): part of the code where an entity can be referenced
  - A binding between a name and the entity is in the environment
- Lifetime of an entity: time interval in which the entity exists
- There must be a precise relationship between lifetime and scope
  - If an entity is destroyed when it is in scope, dangling reference!

Rust and Dangling References

- Rust avoids dangling references/pointers by destroying a value when it is not owned by any variable
  - The variable owning it goes out of scope
  - The (mutable) variable is assigned a new value
- However, a value can be borrowed (references)...
  - The compiler must ensure that a value is not destroyed when it is borrowed
  - Restrictions on borrowing/references
- How can the compiler enforce this?
- Other way to see it: references are variables $\rightarrow$ every reference has a lifetime
  - End of value lifetime $>$ end of reference lifetime

Enforcing the Validity of References

- For nested blocks, it is simple
- What about function invocations?
  - Passing a reference as an argument to a function $\rightarrow$ borrowing
  - The function might return such a reference...
  - ...And in some cases the borrow checker might be in trouble!
- Some help is needed $\Rightarrow$ explicit lifetime annotations
  - Lifetime: similar to a type parameter
  - Lowercase and starting with `'`
  - Example: "`fn f<'a,'b>(v: &'a i32, b: &'b i32) -> &'a i32`"
- The issue exists for custom data types too...

Lifetime Annotations

- When are lifetime annotations needed?
  - In general, every time a function returns a reference...
  - ...Or every time a custom type contains/wraps a reference
- Can lifetime annotations be omitted some times?
  - When the lifetime of the return value (or wrapped/contained) reference is univoque, the compiler can infer it
  - Lifetime inference $\rightarrow$ lifetime elision

```
fn f(v: (&i32, i32)) -> &i32 {
  let (a, b) = v;
  a
}
```