

Advanced Data Types in Rust

Luca Abeni

`luca.abeni@santannapisa.it`

March 24, 2023

Compound Types and Custom Types

- Compound types: tuples and arrays
 - Products of other types
 - Array: all elements have the same type; elements accessible through an index
- Custom types: structures and enumerations
 - Products and sums (**disjoint** unions) of other types
 - Allow to *define new types* and give them a name
- A value of a compound or custom type is composed by multiple elements
 - How to access the single elements?
 - *Destructure* the type, or *unwrap* the contained values

Arrays

- Collections of n elements of the same type T : “[T ; n]”
 - Random access is possible; out-of-bound accesses are checked
 - The array size n is part of the type: possible checks at build time and at runtime
- Dynamically sized view of an array: slice
 - A slice can be seen as a reference “& [T]”
 - The slice size is stored somewhere in a “fat pointer” data structure
- Slices are initialized from arrays: whole array (`let s = &v[..]`) or part of an array (`let s = &v[2..6]`)

Tuples

- Products of different types
- Elements can be accessed through pattern matching
 - Destructuring the tuple
 - No accesses through index variables
- Special “structure-like” syntax

```
let t = ("Hi!", 2);  
let (v1, v2) = t;  
  
println!("{}", v1, v2);  
println!("{}", t.0, t.1);
```

Structures

- Product type, with a name
- Difference with tuples: each field has a name
- Fields accessible through their names, using C-like dot notation
- Can also be quickly destructured using pattern matching

```
struct Point {  
    x: f32,  
    y: f32,  
    z: f32  
}
```

```
let s = Point {x: 1.0, y: 1.0, z: 1.0};
```

```
let Point {x: x1, y: y1, z: z1} = s;
```

```
println! ("{}_{}_{}", x1, y1, z1);
```

```
println! ("{}_{}_{}", s.x, s.y, s.z)
```

Strange Structures

- Unit structure: no fields
 - Unit-like type (type with a single value) with a name
 - `“struct EmptyStruct;”`
 - Will be useful for building enumerations (variants with single value)
- Tuple-like structures
 - Again, will be useful for building enumerations
 - `“let s = StrangeStruct(1.0, 2.0, 3.0);”`
 - Pattern matching will be useful here, too

Enumerations

- Sum type, with a name
- Comma-separated list of structures
 - These structures are the constructors
 - Unit structures: constructors with no argument (C enums)
 - Tuple structures: constructors with arguments (an argument per field); C unions
 - C-like structures can be used too, but are less useful
- Pattern matching is the only way to destructure them/unwrap the contained values
 - For enumerations, there is no other way to access the type elements

Pattern Matching

- Rust provides a powerful pattern matching mechanism, that can be used to:
 - Implement “case-like” switches
 - Define variables, assigning values to them
 - Destructure complex data types
 - Unwrap values contained in algebraic data types
- Pattern matching is used in various constructs, such as:
 - `match`, `if let`, and similar
 - Variables definitions (`let` statements)
 - Parameters passing

Rust Patterns

- A pattern can be:
 - A value (literal, constant)
 - A variable
 - A compound or custom type (tuple, structure, enumeration, ...)
 - The “wildcard pattern”
- A pattern is “matched” by comparing it with some value
 - A constant/literal obviously matches with its value
 - A variable matches with any value of the same type
 - Example: `let pi = 3.14`

More Complex Matching Rules

- A compound/custom value matches if all the elements match
 - For tuples, it is simple:

```
let t = ("Hi!", 2);  
let (v1, v2) = t;
```

 - Here, “(v1, v2)” matches “(“Hi!”, 2)” because “v1” matches ““Hi!”” and because “v2” matches “2”