Using data types

Jonathan Dönszelmann & Vivian Roest

Delft University of Technology

2024-11-18

TU Delft

Lectures about programming are complicated

- One of the best ways to learn is by doing
  - Labs
- You need to learn enough to be able to start
- How to get you to a starting point as quickly as possible?
- Order matters

Data Types in Rust

Question:

What even is a Data type?

Data Types in Rust

Question:

What even is a Data type?

Does python have data types?

Data Types in Rust

Question:

What even is a Data type?

Does python have data types? Let’s discover data types from the ground up!

Theoretical Computer Science

A Hierarchy of computational power

Theoretical Computer Science

A Hierarchy of computational power

- Finite state machines
  - Limited memory, only the current state
  - Limited computation

Theoretical Computer Science

A Hierarchy of computational power

- Finite state machines
- Push down atomata (or context-free languages)
  - Memory bound to rules
  - Memory can affect computations in a limited way

Theoretical Computer Science

A Hierarchy of computational power

- Finite state machines
- Push down atomata (or context-free languages)
- Turing machines
  - Infinite tape of memory
  - Memory can affect computations

Theoretical Computer Science

A Hierarchy of computational power

- Finite state machines
- Push down atomata (or context-free languages)
- Turing machines

RAM isn’t so different from an infinite tape of memory

Our first data types: categorizing size

- a `u8` is just one byte
- a `u64` is 8 bytes

```rust
// compiler, when I call `foo` somewhere in my code
fn foo() {
  // please make some room for me to use 4 bytes for something
  // I'll use the name `a` when I want to use it
  let a: u32 = 3;

  // and 16 more here, I'll call it b
  let b: u128 = 100_000;
}
```

Our first data types: categorizing size

- a `u8` is just one byte
- a `u64` is 8 bytes
- very explict: `[u8; 10]` means exactly 10 bytes

```rust
let a: [u8; 8] = [1, 2, 4, 8, 16, 32, 64, 128];
let b: [u8; 8] = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80];
let c: u64     = 0x01_02_04_08_10_20_40_80;
```

Our first data types: categorizing size

- very explict: `[u8; 10]` means exactly 10 bytes

```Rust
let a: [u8; 8] = [1, 2, 4, 8, 16, 32, 64, 128];
let b: [u8; 8] = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80];
let c: u64     = 0x01_02_04_08_10_20_40_80;
```

Question:

are a, b and c the same?

https://godbolt.org/z/G3doh6e4v

Describing grouped values

Different types bundled together, called a “tuple”:

```Rust
let today: (u8, u8, u32) = (7, 9, 2024);
let tomorrow: (u8, u8, u32) = (8, 9, 2024);
```

Describing grouped values

Different types bundled together, called a “tuple”:

```rust
let today: (u8, u8, u32) = (7, 9, 2024);
let tomorrow: (u8, u8, u32) = (8, 9, 2024);
```

Which we can name:

```rust
type Date = (u8, u8, u32);
// ...
let today: Date = (7, 9, 2024);
let tomorrow: Date = (7, 9, 2024);
```

Describing grouped values

Or a more common way to write that:

```Rust
struct Date {
  day: u8,
  month: u8,
  year: u32,
}
// ...
let today: Date = Date {
  day: 7,
  year: 2024,
  month: 9,
};
```

Describing grouped values

```rust
struct Date {
    day: u8,
    month: u8,
    year: u32,
}
```

Question:

How many bytes do we need for a `Date`?

https://godbolt.org/z/jY9WMPvhj

Struct layout

- Rust has lots of freedom with struct layouts
- https://doc.rust-lang.org/reference/type-layout.html
- Optimized code can take advantage of this

Examples of unaligned accesses: https://godbolt.org/z/8xGaEeqr6

Assignment: 5 minutes

Go to play.rust-lang.org and define a `struct UdpHeader` with these fields:

IPv6 pseudo header format

Offsets Octet		0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source IPv6 Address																															
4	32																																
8	64																																
12	96																																
16	128	Destination IPv6 Address																															
20	160																																
24	192																																
28	224																																
32	256	UDP Length																															
36	288	Zeroes																								Next Header = Protocol[14]							
40	320	Source Port																Destination Port															
44	352	Length																Checksum															

Function Signatures

- Communicates behavior of code through types

```rust
fn is_even(value: i64) -> bool {...}
fn contains(haystack: &[i64], needle: i64) -> bool {...}

// can you guess the name?
fn ______(haystack: &[i64], needle: i64) -> usize {...}
```

Function Signatures

- Communicates behavior of code through types

```Rust
fn is_even(value: i64) -> bool {...}
fn contains(haystack: &[i64], needle: i64) -> bool {...}

// can you guess the name?
type Index = usize;
fn ______(haystack: &[i64], needle: i64) -> Index {...}
```

Types and Behaviors

Question:

What's the difference between `u64`, `i64`, and `f64`

Question:

Why is there a difference?

Types and Behaviors

- Types (mostly*) disappear at runtime
- But types do influence what code is generated

Moving 64 bits around: https://godbolt.org/z/1KshYfYxK

Dividing floats and integers: https://godbolt.org/z/d9ofb1YvK

Adding a behavior to a type

- The `impl` keyword

```Rust
struct SomeType;

impl SomeType {
  fn do_something(&self) {}
}
```

Adding a behavior to a type

- The `impl` keyword
- For example:

```rust
// somewehere in the standard library
struct u64;

impl u64 {
  fn add(&self, other: Self) -> u64 {}
}
```

Adding a behavior to a type

- The `impl` keyword
- For example:

```Rust
// somewehere in the standard library
struct u64;

impl u64 {
  fn add(&self, other: Self) -> u64 {}
  // actualy roughly means:
  fn add(self: u64, other: u64) -> u64 {}
}
```

Adding a behavior to a type

- The `impl` keyword
- Now we can:

```rust
// somewhere in the standard library
impl u64 {
  fn add(&self, other: Self) -> u64 {}
}

let x: u64 = 3;

x.add(5)
```

Adding a behavior to a type

- The `impl` keyword
- Now we can:

```Rust
// somewhere in the standard library
impl u64 {
  fn add(&self, other: Self) -> u64 {}
}

let x: u64 = 3;

x.add(5)
// but you might be more used to:
x + 5
```

Examples of Types that add behaviors

Wrapping integer types

```Rust
use std::num::Wrapping;

let zero = Wrapping(0u32);
let one = Wrapping(1u32);

assert_eq!(u32::MAX, (zero - one).0);
```

(from: https://doc.rust-lang.org/stable/std/num/struct.Wrapping.html)

Examples of Types that add behaviors

- `Saturating<T>`
- references: `&T` and `&mut T` (more next lecture!)
- `std::fs::File`, represents a file that you can interact with

Assignment: 5 minutes

Go to play.rust-lang.org:

- Create a struct called `Range` with two integer fields, `start` and `end`
- Add these functions to the `Range` type
  - `len` which says how far `start` is from `end`
  - `middle` which gives the middle of the range
  - `new` which creates a new range and checks whether `end` > `start`

Hint: use `assert!(a > b)` to make sure conditions hold (and crash otherwise)

Examples of Types that add behaviors

- `Saturating<T>`
- references: `&T` and `&mut T` (more next lecture!)
- `Vec<T>`: dynamically sized array

Processes and Memory

- Like an infinite tape of cells
- Types can give structure

Processes and Memory

- Like an infinite tape of cells
- Types can give structure
- The operating system already gives us some structure
  - The Heap
  - The Stack

Processes and Memory

- Like an infinite tape of cells
- Types can give structure
- The operating system already gives us some structure
  - The Heap
  - The Stack

```c
int* heap_ptr = malloc(sizeof(int));
*heap_ptr = 8;

// don't forget to free!
free(heap_ptr)
```

Examples of Types that add behaviors

- `Saturating<T>`
- references: `&T` and `&mut T` (more next lecture!)
- `Box<T>`: pointer to the heap

```Rust
// NOTE: pseudocode
impl Box<T> {
  pub fn new(value: T) -> Self {
    let pointer = malloc(size_of::<T>());
    // write the value there
    *pointer = value;
    // return a wrapped pointer
    return Box(pointer);
  }
}
```

Examples of Types that add behaviors

- `Saturating<T>`
- references: `&T` and `&mut T` (more next lecture!)
- `Box<T>`: pointer to the heap

C

```c
int main() {
  int* heap_ptr = malloc(sizeof(int));
  *heap_ptr = 8;
  // don't forget to free!
  free(heap_ptr)
}
```

Becomes

Rust

```rust
fn main() {
  let heap_ptr = Box::new(8);
}
```

Examples of Types that add behaviors

- `Saturating<T>`
- references: `&T` and `&mut T` (more next lecture!)
- `Box<T>`: pointer to the heap

```rust
// NOTE: pseudocode
impl Box<T> {
  pub fn drop(&mut self) {
    // automatically frees
    free(self.pointer);
  }
}
```

Examples of Types that add behaviors

- `Vec<T>`: growable collection on the heap

```rust
let mut x = Vec::new();

x.push(3);
x.push(4);
x.push(5);
assert_eq!(x.as_slice(), &[3, 4, 5]);

x.push(6);
assert_eq!(x.as_slice(), &[3, 4, 5, 6]);

x.pop();
assert_eq!(x.as_slice(), &[3, 4, 5]);
```

Types as Proofs

- Types don’t have to be related to memory

Types as Proofs

- Types don't have to be related to memory
- Zero-sized types are even possible:

```Rust
// look ma, no fields!
struct ZeroSized {};

let x: ZeroSized = ZeroSized {};
```

There are usecases for this!

https://www.hardmo.de/article/2021-03-14-zst-proof-types.md

Types as Proofs

- Types don't have to be related to memory
- They can also communicate that you checked something.

```rust
struct Range { start: usize, end: usize }

impl Range {
  pub fn new(start: usize, end: usize) -> Self {
    assert!(end > start);
    Self { start, end }
  }

  pub fn len(&self) -> usize {
    self.end - self.start // cannot fail! (can't be negative)
  }
}
```

Types as Proofs

- Types don’t have to be related to memory
- They can also communicate that you checked something.
- Examples:
  - `NonZero<T>` proves that the integer `T` is not zero

Strings

- Array of bytes
- Certain bytes mean certain characters

```Rust
let a = "hello";
```

- You might know about ASCII

Strings

- Array of bytes
- Certain bytes mean certain characters

```rust
fn first_char(input: &str) -> char {
  input.chars().next().expect("must have at least one character")
}

assert_eq!(first_char("مرحبًا!"),'م');
```

- Unicode aims to include everyone!

Types as Proofs

- Types don't have to be related to memory
- They can also communicate that you checked something.
- Examples:
  - `NonZero<T>` proves that the integer `T` is not zero
  - `&str` is like bytes (`&[u8]`), but proves UTF-8
  - `String` is like a `Vec<u8>` but proves UTF-8
  - `*const u8` doesn't prove the `u8` is valid, `&u8` does

Assignment: 5 minutes

- Create a type called `Even`
- Create a `new` method that checks this property
- Create a `divide_by_two` method that never has to round
- Create a `multiply` method that:
  multiplies the even number
  by another number (even or not)
  and produces a new even number

Individual Assignment

- Graded for 50% of your grade
- DSMR Telegram parser
  - See https://cese.ewi.tudelft.nl for all info
- An assignment to get you familiar with the basics of Rust
- Don’t be scared about the sheer amount of documentation online, take it step by step
- Ask questions in the labs