3 Compound Data Types

Thomas Prokosch
October 30, 2018

Outline

Sum types and product types

Rust supports algebraic data types which are compound data types. Assume there are types $T_1, T_2, \ldots, T_n$.

- A sum type may contain any of these types but **only one concurrently**.
- A product type must contain **all types concurrently**.

type	example	nr of values
Σ sum	Bool = False $\mid$ True	$1 + 1$
	Trool = False $\mid$ True $\mid$ DontKnow	$1 + 1 + 1$
Π product	any tuple, e.g.: (u8, u8)	256×256
	(u8, bool, i32)	$2^8 \times 2 \times 2^{32}$
	unit type ()	1
	(u8, ())	$2^8 \times 1$

Structures with `struct`

Usage of `struct`

To create a product type:

- **Declaration** of a `struct`

  ```
  struct Point {x: i32, y: i32}
  ```

  Note the **missing semicolon**!

- **Definition** of `struct`

  ```
  let origin = Point {x:  0, y:  0};
  let mut p  = Point {x: 10, y: 11};
  ```

- **Element access**

  ```
  let px: i32 = p.x;
  p.x = 5;
  ```

Field init shorthand for `struct`

Variable name identical to **field name**: Write `x` instead of `x: x`.

```
struct User { uid: String, email: String, signin_ct: u64 }
fn new_user(uid: String, email: String) -> User {
    User { uid, email, signin_ct: 0 }
}                                  // ALTERNATIVE: uid: uid
fn main() {
    let to_str = String::from;
    let u1 = User {
        email: String::from("frizzyhair@rust-lang.org"),
        uid: String::from("frizzyhair"),
        signin_ct: 1
    };
    let u2 = new_user(to_str("fuzzbuzz@rust-lang.org"),
                      to_str("fuzzbuzz"));
}
```

Functional updates

Update multiple values using struct update syntax from the same (or another) struct:

```
let user2 = User {
    email: String::from("squit@rust-lang.org"),
    username: String::from("squit"),
    ..user1
};
```

`..` means that remaining fields (not explicitly set) get the **same values** as in the **given instance**

Tuples

- unnamed `struct` (a structure **without component names**)
- **access:** via pattern matching `(x, y, z)` or direct access `.0`, `.1`, etc.
- **singleton tuple:** `(x, )`
  `(x)` indistinguishable from parenthesized value
- **unit type** is empty tuple: `()`

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
let x_axis = origin.0;
```

`Color` and `Point` are different types, **cannot be assigned** mutually!

Pattern matching of tuples

Tuples can be **destructured** using pattern matching:

```
struct Point(i32, i32);

let p = Point(0, 8);
match p {
    Point(x, 0) => println!("On the x axis at {}", x),
    Point(0, y) => println!("On the y axis at {}", y),
    Point(x, y) => println!("On neither axis: ({},{})", x, y)
}
```

yields

```
On the y axis at 8
```

Type aliasing

Type aliasing: A new type that can be used interchangeably with the original type, to **aid code understanding**.

```
type Kilometers = i32;
type Miles      = i32;
let  distance: Kilometers = 5;
```

However, the following is possible:

```
let mut marathon: Miles = 26;
marathon = distance;
```

Solution: Newtype pattern (origin: Haskell)

Newtype pattern

Use the newtype pattern to create a

- **unique type** with **hidden implementation** and
- no runtime performance penalty.

```
struct Kilometers(i32);
struct Miles(i32);
let    distance = Kilometers(5);
let    marathon: Miles = distance; // IMPOSSIBLE
```

These are **tuples**, so access the value like that:

```
println!("You walked {}km.", distance.0);
```

Enumerations with `enum`

Usage of `enum`

To create a sum type:

```
enum Animal {
    Dog,
    Cat,
}
```

(An animal is either a dog, or a cat — but not both!)

```
let mut a: Animal = Animal::Dog;
a = Animal::Cat;
```

- **Declaration** with `enum`, then **enumerating** the alternatives
- **Definition** using the `enum` name, the path separator `::`, and the variant name

enum with fields

An `enum` may also contain fields (using the `struct` or tuple syntax):

```
enum Animal {
    // enum variant
    Dog(String, f64),
    // struct-like enum variant
    Cat { name: String, weight: f64 },
}

let mut a: Animal;
a = Animal::Dog("Snoopy".to_string(), 5.6);
a = Animal::Cat { name: "Garfield".to_string(),
                  weight: 37.2 };
```

Variants may have **different fields**.

Methods with `impl`

Method syntax in Rust

Rust allows to define methods on data types:

```
let s  = String::from("Welcome to third lecture");
let l0 = String::len(&s);
let l1 = s.len();
```

where the signatures looks roughly like this:

- `fn from(s: &str) -> String`
  `from` is a String-associated function which **does not operate** on a String itself, hence the call convention with `::`
- `fn len(&self) -> usize`
  `&self` is a special reference always **referring to the data item** for which the function is called. `len` also is a String-associated function and can either be called with the `::` or the dot call convention.

Defining methods

Methods for a data type implemented using `impl` keyword:

```
struct Color(f32, f32, f32);

impl Color {
    fn average(&self, c: &Color) -> Color {
        Color((self.0 + c.0)/2.,
              (self.1 + c.1)/2.,
              (self.2 + c.2)/2.)
    }
}

let cyan = Color(0.0, 1.0, 1.0);
let blue = Color(0.0, 0.0, 1.0);
let avg  = cyan.average(&blue);
println!("average is: ({},{},{})", avg.0, avg.1, avg.2);
```

Error handling

Option<T>

Data types may be **parametrized using type variables**:

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

This type constructor allows to store data of any type `T`, or `None` if data unavailable.

language	equivalent
C++17	std::optional<T>
C#	Nullable<T>
Haskell	Maybe a
Java	Optional<T>
Julia	Nullable{T}
Scala	Option[A]
Swift	Optional<T>

`Option<T>` defines several methods, for example `fn unwrap_or` allows to give a **default value**:

```
fn list_dir(dir: Option<std::path::Path>) {
    let it = dir.unwrap_or(Path::new("./")).read_dir();
    // more work needed ...
```

`Result<Ok, Err>`

`Result<T, E>` provides **detailed failure information**:

- `Ok(T)`: element of type `T` was found
- `Err(E)`: an error of type `E` was encountered

```
match "15_".parse::<u32>() {
    Err(e) => println!("Error: {}", e),
    Ok(i)  => println!("Parsed: {}", i)
}
```

yields

```
Error: invalid digit found in string
```

`parse` **returns a generic type**, so a type annotation is required. This is done using the turbofish syntax `::<>`

Example: Read file into string

```
use std::io::Read;
fn readf(path: &str) -> Result<String, &'static str> {
    if let Ok(mut file) = std::fs::File::open(path) {
        let mut buf = String::new();
        match file.read_to_string(&mut buf) {
            Ok(_) => Ok(buf),
            _     => Err("Error while reading")
        }
    } else {
        return Err("Cannot open file")
    }
}
match readf("/etc/os-release") {
    Ok(f)  => println!("Got: {}", f),
    Err(e) => println!("Err: {}", e)
}
```

The `?` operator

Common pattern:

- Do something on `Ok()`
- Return immediately on `Err()`

Solution: short-cut evaluation using `?`

`?` can even be used **inside expressions**:

```
use std::io::Read;
fn readf(path: &str) -> Result<String, std::io::Error> {
    let mut buf = String::new();
    std::fs::File::open(path)?
        .read_to_string(&mut buf)?;
    Ok(buf)
}
match readf("/etc/os-release") {
    Ok(f)  => println!("Got: {}", f),
    Err(e) => println!("Err: {}", e)
}
```

The `panic!` macro

Unconditionally abort program execution:

```
fn main() {
    panic!("crash and burn");
}
```

No way to recover, use for:

- development purposes (`panic!` **compatible with any type**)
- getting a program backtrace
- examples, prototypes, tests
- nothing else

Some functions panic, e.g.: `unwrap()`, use only if you are **smarter than Rust**:

```
use std::net::IpAddr;
let home: IpAddr = "127.0.0.1".parse().unwrap();
```

Recursive types

Dynamically Sized Types (DST)

Variables and function parameters

- are stack allocated and
- must have known size (at compile time).

Dynamically Sized Types (DST) — Examples:

- **slices** — cannot be assigned to variables without references, e.g. `str`
- **recursive types** have no immediate representation:

  ```
  enum List<T> {
      Nil,
      Cons(T, List<T>)
  }
  ```

  This recursive definition has no known size, results in a **compile-time error**.

Solution: Use references/pointers and heap allocation.

Boxed values

Heap is a memory region which

- is decoupled from functions
- **dynamically** provides arbitrarily-sized memory chunks
- has no inherent size limit
- uses **random access** (slower!)
- is used with pointers

`Box<T>`

- Common use-case of pointers: Point to some object (on the heap)
- `Box` **is a container** for objects `T` on the heap

```
let b = Box::new(5);          // allocate on heap
println!("I've got {}", *b); // operator * dereferences
println!("Me, too: {}",  b); // automatic dereference
```

No deallocation necessary, Rust takes care of it.

Dynamically sized types with Box

```
enum List<T> {
    Nil,
    Cons(T, Box<List<T>>)
}
impl List<i32> {
    fn new() -> List<i32> { List::Nil }
    fn print(&self) {
        match &self {
            List::Nil => {
                println!(""); },
            List::Cons(a, l) => {
                print!("{}, ", a); l.print(); }}}
    fn add_front(self, v: i32) -> List<i32> {
        List::Cons(v, Box::new(self)) }
}
List::new().add_front(5).add_front(7).print();
```

Zero Sized Types (ZST)

Zero Sized Types (ZST)

Zero-sized types (ZST) do not require space:

- type with **no values** — nothing to store
- types with a **single value** — is constant, can be optimized away

No values: The Void / Never type

- cannot be instantiated (there are no values)
- must not be confused with the `void` type from C/C++ which contains a single element `void`, equivalent to unit type `()`
- cf. Haskell's `Data.Void`

`struct VoidS;` `enum VoidE{};` `!` (predefined, experimental)

Diverging functions

```
fn panic()  -> ! { ... }
fn server() -> ! { ... }
```

Expressing non-termination allows some **optimizations** w.r.t. the unit `()` return value

Abandon pre-defined code paths

```
enum Pipe<A, B, R> {
    Value(R),
    Await(fn (A) -> Pipe<A, B, R>),
    Yield(B, Box<Pipe<A, B, R>>) }
```

to get a `Pipe` that never yields:

```
struct Void;
type Consumer<A, R>
    = Pipe<A, Void, R>;
```

Zero sized types with one value

Any operation that produces or stores a **ZST can be reduced to a no-op**.

```
struct Foo; // No fields = no values

// All fields have no size = no size
struct Baz {
    foo: Foo,
    qux: (),      // empty tuple has no size
    baz: [u8; 0], // empty array has no size
}
```

Sets are maps with unit `()` values.

- `()` values get removed from the resulting binary, do **not slow down**
- one code base, less bugs

→ zero cost abstraction

```
struct HashMap<K, V, S = RandomState> {
    // some fields omitted
}
type HashSet<K, S = RandomState> = HashMap<K, (), S>;
```

Void vs. `()`

Why does Rust use `()` instead of `!` (the never type)?

1. The never type `!` is not yet finalized.
2. Both `!` and `()` do get optimized away, so they are equal.
3. When using `!` you cannot put elements into the set.
4. Returning `()` is simplified if the element look-up fails.

Zero-sized types in Rust

Sets are maps with unit `()` values.

- `()` values get removed from the resulting binary, do **not slow down**
- one code base, less bugs

→ zero cost abstraction

```
struct HashMap<K, V, S = RandomState> {
    // some fields omitted
}
type HashSet<K, S = RandomState> = HashMap<K, (), S>;
```

Void vs. `()`

Why does Rust use `()` instead of `!` (the never type)?

3 When using `!` you cannot put elements into the set.

Collections

Overview

Collection types in **Rust's standard library**:

Sequences	Vec, VecDeque, LinkedList, BitVec
Maps	HashMap, BTreeMap, VecMap
Sets	HashSet, BTreeSet, BitSet

	Vec	HashMap	BTreeMap	HashSet	BTreeSet
retains order	**yes**	no	no	no	no
sorted by keys	no	no	**yes**	no	**yes**
retrieve	**O(1)**	**O(1)**	O(log n)	**O(1)**	O(log n)
insert	at end	**O(1)**	O(log n)	**O(1)**	O(log n)

The other collection types cover **special use-cases**, not discussed here

Iterators

- provide sequences of values

generic	implementable for many types, including custom types
safe	iterator does not allow modification of collection
efficient	lazy evaluation (calculation only on demand)

- consumed using a **for loop**
- can be chained and combined

iter	**immutable** references	perform calculation
iter_mut	**mutable** references	update collection
into_iter	iterator **consumes** collection	transform collection

Iterator examples

- `fn iter()`:

```
let vec = vec![1, 2, 3, 4];
for x in vec.iter() {       // same as: &vec
    println!("vec contains {}", x);
}
```

- `fn iter_mut()`:

```
let mut vec = vec![1, 2, 3, 4];
for x in vec.iter_mut() {   // same as: &mut vec
    *x += 1;
}
```

Use $*$ **dereference operator** to retrieve element from iterator reference. Mandatory for assignments, optional for reads.

Converting collections with `into_iter()`

Use `fn into_iter()` and `fn collect()` together with **explicit type annotation** of the target collection:

```
let vec = vec![1, 2, 3, 4];
let set: std::collections::BTreeSet<i32>
    = vec.into_iter().collect();
// vec consumed, no longer accessible
println!("Set contains 2: {}", set.contains(&2));
println!("Set contains 5: {}", set.contains(&5));
```

```
Set contains 2: true
Set contains 5: false
```

Lazy evaluation and iterator chaining

Print the sum of all squares which are less than 10.

```
let mut result = 0;
for i in (0..).map(|x| x*x).take_while(|x| *x < 10) {
    result += i
}
println!("Sum(squares<10) = {}", result);
```

```
Sum(squares<10) = 14
```

Hash maps/sets

- associate **arbitrary keys** with an arbitrary value
- keys/values are in **no specific order**
- insertion and look-up are **fast: O(1)**
- grow/shrink on demand
- iterator returns tuple of keys and values

```
let mut contacts = std::collections::HashMap::new();
contacts.insert("Daniel", "798-1364");
contacts.insert("Ashley", "645-7689");
contacts.insert("Katie",  "435-8291");
contacts.insert("Robert", "956-1745");

match contacts.get(&"Susan") {
    Some(&number) => println!("Calling Susan: {}", number),
    _ => println!("Don’t have Susan’s number.") }
for (contact, &number) in contacts.iter() {
    println!("Calling {}: {}", contact, number); }
```

BTree maps/sets

- **balanced tree**, may have degree greater 2
- B-trees keep **data sorted**
- most operations in **logarithmic time**
- suited for databases/disks with **large amounts** of data

```
let mut count = std::collections::btree_map::BTreeMap::new();
let message = "she sells sea shells by the sea shore";

for c in message.chars() {
    *count.entry(c).or_insert(0) += 1;
}

println!("Number of occurrences of each character");
for (char, count) in count.iter() {
    println!("{}: {}", char, count);
}
```

Destructuring data with pattern matching

Where can patterns occur?

match arms	`match VALUE { PATTERN => EXPR }`	irrefutable
if let	`if let PATTERN = EXPR { }`	refutable
while let	`while let PATTERN = EXPR { }`	refutable
for loops	`for PATTERN in EXPR { }`	irrefutable
let statements	`let PATTERN in EXPR;`	irrefutable
fn parameters	`fn (PATTERN: TYPE) -> TYPE { }`	irrefutable

- refutable: pattern **can fail** to match
- irrefutable: pattern **must not fail** to match for any value

Pattern matching syntax

- **multiple patterns:** `v1 | v2` matches both v_1 and v_2
- **ranges:** `...` is the same as `..=`, exclusive range `..` not yet available
- **ignoring a single value:** underscore _ ignores (part of) a value
- **ignoring multiple values:** `..` ignores following elements
  may be used in the middle like so: `(first, .., last)`
- **match guards:** `p if boolexpr` only matches pattern *p*
  if `boolexpr` is true
- `@` **bindings:** `x @ p` matches pattern *p* but also associates variable *x*
  with whole expression

```
enum Message { Quit, Write { s: String, b: bool } }
let msg = Message::Write {
    s: "Lemons are green".to_string(), b: false };
match msg {
    Message::Write { b: true, s: x } => println!("{}", x),
    Message::Write { s, .. } => println!("Not true: {}", s),
    Message::Quit => println!("See you!") }
```

Pattern matching example

Order of patterns is important: **First arm** is taken.

```
for x in vec![4, 3, 100, -1, 0] {
    match x {
        n @ 1..=10 if n%3 == 0 =>
            println!("A small number divisible by 3"),
        4 | 8    => println!("My favourite number"),
        50...100 => println!("A big number"),
        n if n<0 => println!("Don't be so negative!"),
        _        => println!("What are you up to?")
    }
}
```

Summary

Today's goals:

1. Be able to create sized and recursive data types.
2. Learn how to implement methods.
3. Start using the heap by creating pointers to values.

We have learned about...

- sum and product types
- methods
- recursive types and boxed values
- collections and iterators
- type aliasing and the newtype pattern
- error handling and pattern matching in detail