CMSC 330: Organization of Programming Languages

Structs, Enums in Rust

Slide credit: Michael Hicks, Niki Vazou

Rust Data

- So far, we've seen the following kinds of data
  - Scalar types (int, float, char, string, bool)
  - Tuples, Arrays, and Collections
- How can we build other data structures?
  - Structs (like Objects; support for methods)
  - Traits (like Interfaces)
  - Enums (like Ocaml Data Types)

Structs: Definitions & Construction

```
struct Rectangle {
  width: u32,
  height: u32,
}

impl Rectangle {
    fn new(width: u32, height: u32) -> Rectangle {
        Rectangle {width, height}
    }
}
fn main() {
  // construction
  let rect1 = Rectangle::new(30, 50);
  // accessing fields
  println!("rect1's width is {}", rect1.width);
}
```

abbreviation for {width: width, height: height}

> rect1's width is 30

Structs: Printing

```
struct Rectangle {
  width:u32,
  height:u32,
}

fn main() {
  let rect1 = Rectangle::new(30, 50);
  println!("rect1 is {}", rect1);
}
```

```
error[E0277]: the trait bound `Rectangle:
std::fmt::Display` is not satisfied
```

Structs: Printing with Derived Traits

```
#[derive(Debug)]

struct Rectangle{
  width:u32,
  height:u32,
}

fn main() {
  letrect1 = Rectangle::new(30, 50);
  println!("rect1 is {:?}", rect1);
}
```

Derive debug printing format

Use debug printing format

> rect1 is Rectangle { width: 30, height: 50 }

Structs

- Syntax
  - `struct` *T* [<*T*>] {*n1*:*t1*, …, *ni*:*ti*,}
  - the *ni* are called fields, begin with a lowercase letter
  - [<*T*>] optionally for generics (see later)
- Evaluation
  - **Construction:** *T{n1:v1, ni:vi}* is a value if **vi** are values.
  - **Projection:** **t.ni** returns the **ni** field of **t**
- Type Checking
  - *T{n1:v1, ni:vi}* : *T* [if *vi* has type *ti*]

Quiz 1: `point` is immutable at *HERE*

```
struct Point {
  x: i32,
  y: i32,
}
let mut point = Point { x: 0, y: 0 };
point.x = 5;
let point = point;
// HERE
```

A. True
B. False

Quiz 1: `point` is immutable at *HERE*

```
struct Point {
  x: i32,
  y: i32,
}
let mut point = Point { x: 0, y: 0 };
point.x = 5;
let point = point;
// HERE
```

A. True
B. False

Mutability is a property of the binding; the old point's contents are moved to the new one

A note on mutability

- A failed attempt to make a `Point` that is always mutable:

```
struct MutablePoint {
  x: mut i32,
  y: mut i32,
}
```

```
error: expected type, found keyword `mut` --> src/main.rs:2:6
  |
2 | x: mut i32,
  |    ^^^ expected type
```

- This code attempts to make mutability part of the type declaration, but mutability is a property of the variable that holds the MutablePoint.

Methods: Definitions on Structs

Self argument has type Rectangle

```
impl Rectangle {
  fn area(&self) -> u32 {
    self.width * self.height
  }
}
```

Self argument is a borrowed **reference** to the object

impl Rectangle defines an implementation block

- self arg has type Rectangle (reference)
- ownership rules:
  - &self for read-only borrowed reference (preferred)
  - &mut self for read/write borrowed reference (if needed)
  - self for full ownership (least preferred, most powerful)

Methods: Calls

```
fn main() {
  let rect1 = Rectangle::new(30, 50);
  println!("The area is {} pixels.", rect1.area());
}
```

dot syntax to call methods

If method had arguments, use function call e.g.,
rect1.area(3)

Methods: Many Args, Associated Methods

```
impl Rectangle {
 fn can_hold(&self, other: &Rectangle) -> bool {
   self.width > other.width && self.height >
other.height
 }

 fn square(size:u32) -> Rectangle {
   Rectangle { width: size, height: size }
 }
}
```

A **reference** to the Rectangle; most flexible

square is called an associated method

- no `self` argument
- operates on Rectangles
- called with `let sq = Rectangle::square(3);`

Generic Lifetimes

```
struct ImportantExcerpt<'a> {
  part: &'a str,
}

fn main() {
  let novel = String::from("Generic Lifetime");
  let i = ImportantExcerpt { part: &novel; }
}
```

When structs defined to hold **references**, we need to add a lifetime annotation on the reference (here, 'a)

Lifetime is inferred for `i`, above

Note that struct fields can be `&mut` references (although they cannot be mutable themselves)

Lifetimes in Implementation Methods

```
struct ImportantExcerpt<'a> {
  part: &'a str,
}

impl<'a> ImportantExcerpt<'a> {
  fn level(&self) -> i32 {
    3
  }
}
```

Parameter for lifetime annotation
(would need the same for a generic
Implementation of a generic interface in Java)
Sometimes can be inferred ("elision")

Enums

```
enum IpAddr{
  V4(String),
  V6(String),
}

let home      = IpAddr::V4(String::from("127.0.0.1"));
let loopback  = IpAddr::V6(String::from("::1"));
```

definition

construction

Enums with impl blocks

```
enum IpAddr{
  V4(String),
  V6(String),
}

impl IpAddr {
  fn call(&self) {
  // method body would be defined here
  }
}

let m = IpAddr ::V6(String::from("::1"));
m.call();
```

Enums with Structs

Enums might contain any type,
e.g., structs, references, …

```
struct Ipv4Addr{
  // details elided
}

struct Ipv6Addr{
  // details elided
}

enumIpAddr{
  V4(Ipv4Addr),
  V6(Ipv6Addr),
}
```

The Option Enum: Generic Types

Defined in standard lib

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

let some_number = Some(5);
let some_string = Some("a string");
let absent_number:Option<&Rectangle> = None;
```

Instantiation with any type!

Generics in Structs & Methods

Generic T in struct

```
struct
Point<T> {
  x: T,
  y: T,
}
```

Generic T in methods

```
impl<T> Point<T> {
  fn x(&self) -> &T {
    &self.x
  }
}
```

Instantiate T as i32

```
fn main() {
  let p = Point { x:5, y:10};
  println!("p.x = {}", p.x());
}
```

Pattern Matching

- Key feature from functional languages
- Case-analyze on an algebraic data type

```
fn plus_one(x:Option<i32>) -> Option<i32> {
  match x {
    Some(i) => Some(i +1),
    None => None,
  }
}
```

Matching should be exhaustive!

```
fn plus_one(x:Option<i32>) -> Option<i32>
{
   match x {
    Some(i) => Some(i +1),


   }
}
```

Error at compile time!

`error[E0004]: non-exhaustive patterns:`

``None` not covered`

Enums

- Syntax
  - `enum T [<T>] {C1 [(t1)], …, Cn [(tn)],}`
  - the `Ci` are called constructors
    - Must begin with a capital letter; may include associated data notated with brackets [] to indicate it's optional
- Evaluation
  - A constructor `Ci` is a value if it has no assoc. data
    - `Ci(vi)` is a value if it does
  - Eliminating a value of type `t` is by pattern matching
    - patterns are constructors `Ci` with data components, if any
- Type Checking
  - `Ci [(vi)] : T` [if `vi` has type `ti`]

Quiz 2: Output of following code

```
enum Number {
    Zero,
    One,
    Two,
}
use Number::Zero;
let t = Number::One;
match t {
  Zero=> println!("0"),
  Number::One => println!("1"),
}
```

A. 0
B. 1
C. Compile Error

Quiz 2: Output of following code

```
enum Number {
     Zero,
     One,
     Two,
}
use Number::Zero;
let t = Number::One;
match t {
   Zero=> println!("0"),
   Number::One => println!("1"),
}
```

A. 0
B. 1
C. Compile Error. Pattern `Two` not covered

If-let, for non-exhaustive matches

```
fn check(x: Option<i32>) {
  if let Some(42) = x {
    println!("Success!")    // only executed if the match succeeds
  } else {
    println!("Failure!")
  }
}
```

```
fn main (){
  check(Some(3));;   // prints "Failure!"
  check(Some(42));; // prints "Success!"
  check(None);;        // prints  "Failure!"
}
```

Recap: Structs and Enums

1. Structs define data structures with fields
   – And implementation blocks collect methods on to specify the behavior of structs (like objects)

2. Enums define a set of possible data types
   – Use match or if-let to deconstruct

TRAITS

Overview

- Traits abstract behavior that types can have in common
  - Traits are a bit like Java interfaces
  - But we can implement traits over any type, anywhere in the code, not only at the point we define the type

- Trait bounds can be used to specify when a generic type must implement a trait
  - Trait bounds are like Java's bounded type parameters

Defining a Trait

- Here is a trait with a single function

```
pub trait Summarizable {
  fn summary(&self) -> String;
}
```

  - Specify **&self** for "instance" methods
    - Can also specify "associated" methods
      - » Like **static** methods in Java

- Equivalent in Java:

```
public interface Summarizable {
  public String summary();
}
```

Note: The keyword **pub** makes any module, function, or data structure accessible from inside of external modules. The **pub** keyword may also be used in a **use** declaration to re-export an identifier from a namespace.

Note that we make the entire trait public, not individual elements of it.

Implementing a Trait on a Type

name of trait

type on which we are implementing it

```
impl Summarizable for (i32,i32) {
    fn summary(&self) -> String {
        let &(x,y) = self;
        format!("{}",x+y)
    }
}
fn foo() {
    let y = (1,2).summary(); //"3"
    let z = (1,2,3).summary();//fails
}
```

trait method body

trait method invocation

Default Implementations

- Here is a trait with a default implementation

```
pub trait Summarizable {
   fn summary(&self) -> String {
     String::from("none")
   }
}
```

default impl

Impl uses default

```
impl Summarizable for (i32,i32,i32) {}
fn foo() {
    let y = (1,2).summary(); //"3"
    let z = (1,2,3).summary();//"none"
}
```

Trait Bounds

- With generics, you can specify that a type variable must implement a trait

```
pub fn notify<T: Summarizable>(item: T) {
  println!("Breaking news! {}",
              item.summary());
}
```

  - This method works on any type **`T`** that implements the **`Summarizable`** trait
    - This is a kind of subtyping: **`T`** can have many methods but at the least it should implement those in the **`Summarizable`** trait

Trait Bounds: Like Java Bounded Parameters

- Equivalent in Java

```
<T extends Summarizable>
void notify(T item) {
  System.out.println("Breaking news! "+
                     item.summary());
}
```

  - This generic method works on any type `T` that implements the `Summarizable` interface (which we showed before)

```
public interface Summarizable {
  public String summary();
}
```

Generics, Multiple Bounds

- Trait implementations can be generic too

```
pub trait Queue<T> {
    fn enqueue(&mut self, ele: T) -> (); …
}
impl <T> Queue<T> for Vec<T> {
    fn enqueue(&mut self, ele:T) -> () {…} …
}
```

- Generic method implementations of structs and enums can include trait bounds
- Can specify multiple Trait Bounds using **+**

`fn foo<T:Clone + Summarizable>(…) -> i32 {…}` *or*

`fn foo<T>(…) -> i32 where T:Clone + Summarizable {…}`

(Non)Standard Traits

- We have seen several standard traits already
  - `Clone` holds if the object has a `clone()` method
  - `Copy` holds if assignment duplicates the object
    - I.e., no ownership transfer, as with primitive types
  - `Move` holds if assignment moves ownership
    - I.e., because assignment doesn't copy it all; the default
  - `Deref` holds if you can dereference it
    - I.e., it's a primitive reference, or has a **`deref()`** method
- There are other useful ones too
  - `Display` if it can be converted to a string
  - `PartialOrd` if it implements a comparison operator

Note: Several of these traits indicate special treatment by the compiler, e.g., **Move** and **Copy**; they go beyond the indication that an object implements particular methods.

Putting all Together

- Finds the largest element in an array slice
  - Generic in the type **`T`** of the contents of the array

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T
{
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

Requires Copy trait to not transfer ownership

Requires PartialOrd trait

Putting all Together

- Finds the largest element in an array slice
  - Generic in the type **`T`** of the contents of the array

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T
{…}
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);
    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

prints

```
The largest number is 100
The largest char is y
```

Quiz: What is the output

```
trait Trait {
  fn p(&self);
}

impl Trait for u32 {
     fn p(&self) { print!("1"); }
}
let x=100; // inferred as u32
x.p();
```

A. 100
B. 1
C. Error

Quiz: What is the output

```
trait Trait {
  fn p(&self);
}

impl Trait for u32 {
      fn p(&self) { print!("1"); }
}
let x=100; // inferred as u32
x.p();
```

A. 100
B. 1
C. Error