Langages systèmes 2 - Introduction to Rust

Raphael Amiard - Adacore

amiard@adacore.com

AdaCore

Introduction to Rust - Part 2

AdaCore

Pattern matching

```
match e {
    // Matching on a struct inside an enum:
    Expr::BinOp {l, op, r} => ...

    Expr::Literal(1..=5) => ...
    //            ^ Match an integer between 1 and 5 included

    Expr::Literal(8) => ...
    //            ^ Match an integer of value exactly 8

    Expr::Literal(v) => ...
    // Matching on a tuple inside an enum

    Expr::Literal(_) => ...
    // Ignore the value
}
```

Pattern matching

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

// Pattern matching on a struct (kind of useless, gives warning used like that)
match p {
    Point {x, y} => ...
}

// Irrefutable pattern inside let
let Point {x, y} = p

// Will only enter the body of the if if the match worked.
if let Expr::BinOp(l, op, r) = expr {
    ...
}
```

Option type

```
use std::option::Option;
fn get_data() -> Vec<i32> { ...  }

fn main() {
    let a = get_data();
    let val: Option<i32> = a.pop();
    //                     ^ Get last element of vec

    match val {
        Some(val) => println!("Got a value out of vector: {}", val),
        None => println!("No value")
    }
}
```

- **Extremely** common type to represent possibility of a value
- Will be found everywhere

Result type

- Generic type
- Error type (*E*) is often a string (`&'static str`, a static string reference)

```
// Here is how the result type is defined in the stdlib
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

```
fn main() {
    let a = "10".parse::<i32>();

    // Handle either option, error or OK
    match a {
        Ok(val) => println!("{val}")
        Err(e) => println!("No value. Error: {e}")
    }

    // Only handle OK
    if let Some(val) = a {
        println!("{val}")
    }

    println!("{}", a.unwrap_or_else(|| 0));

    // Panic on error
    println!("{}", a.unwrap());
}
```

Pattern matching: loops

```rust
fn get_data() -> Vec<i32> { ... }

fn main() {
    let data = get_data();

    //   Irrefutable pattern
    for (idx, val) in data.iter().enumerate() {
    }

    // Iterate while we can match the pattern
    while let Some(a) = data.pop() {
        ...
    }
}
```

```rust
fn print_point_1(p: (i32, i32)) {
    let (a, b) = p;
    //  ^ This is a pattern
    println!("Current location: ({a}, {b})");
}

fn print_point_2((a, b): (i32, i32)) {
    //          ^ This is a pattern
    println!("Current location: ({a}, {b})");
}
```

Rust traits & generics

```
struct LinkedList<T> {
    item: T,
    next: Box<LinkedList<T>>
}
```

- Like Java/C# generics: abstract over types, functions, not packages
- Like Ada (& others): legality checked in the generic form
- Operations need to be made available on types (via traits)

Generics (2)

```rust
struct HashTable<T> { ... }

impl HashTable<T> {
    fn add(&self, item: T) {
        // problem: how do we hash elements?
    }
}
```

Traits

- Traits define common behavior
- Very similar to interfaces in Java/C#/etc
- But first and foremost a generic concept

```
trait Hashable {
    fn hash() -> i32;
}

struct HashTable<T: Hashable> { }
//                  ^ Trait bound

impl HashTable<T> {
    fn add(&self, item: T) {
        ...
        let hash = item.hash();
        ...
    }
}
```

AdaCore

Shorthand for trait bounds in functions

```rust
fn display_list<T: Display>(list: &[T]) {
    for el in list {
        print!("{el}");
    }
}

// Shorthand:

fn display_list(list: &[impl Display]) ...
// This function is a GENERIC function
```

Some built-in traits

- Rust has a lot of built-in traits that are part of the standard library
- Some of those are derivable: The compiler can provide an implementation for you automatically.
- `Debug`: use to display a value using the `{:?}` formatter
- Ordering traits like `Eq`, `Ord` are used to compare values
- `Copy` and `Clone`, allow different copy semantics for your type.
- `Hash` computes a hash for your type

To derive:

```rust
#[derive(Hash, Debug)]
struct Point {
    x: i32, y: i32
}
// This struct is now hashable and displayable via the Debug trait
```

- The *Clone* trait adds a *clone* function on your type, that allows you to clone an instance of it.
- The *Copy* trait, on the other hand, gives full copy semantics to your type (like you have by default on scalar types).

```rust
#[derive(Copy, Debug)]
struct Point {
    x: i32, y: i32
}

fn main() {
    let p = Point { x = 1, y = 2 };
    let p2 = p;

    println!("{:?}", p);
    // WHAT IS THIS SORCERY
}
```

- You can store any object implementing a trait via the *dyn* qualifier, creating a trait object

```rust
use std::fmt::Debug;

fn main() {
    let a: Vec<Box<dyn Debug>> = vec![
        Box::new(12),
        Box::new("pouet"),
        Box::new((1, 2))
    ];
    println!("{:?}", a);
}
```

Ownership is a combination of three things:

- Basic rules of ownership (one owner, N borrowers, etc)
- Lifetimes for every value. For the moment, all lifetimes were infered.
- The borrow checker: checks that borrows don't outlive the lifetime of the value they borrow

Turns out you can actually specify lifetimes yourself, allowing you to express things that weren't possible before:

```
// Won't work: can't return reference without explicit lifetime
fn smallest (a: &str, b: &str) -> &str {
    if a < b { a } else { b }
}

// Works
fn smallest <'a> (a: &'a str, b: &'a str) -> &'a str {
    if a < b { a } else { b }
}
```

Lifetimes (2)

```
fn smallest <'a> (a: &'a str, b: &'a str) -> &'a str {
    if a < b { a } else { b }
}

fn main() {
    let a = String::from("hello");     // <-| Lifetime for a
    let c;                             //   |
    {                                  //   |
        let b = String::from("world"); //   | <-| Lifetime for b (and hence for c)
        c = smallest(&b, &a);          //   |   |
        println!("{}", c);             //   | <-|
    }                                  //   |
    println!("{}", c);                 // <--
}
```

AdaCore

- Lifetimes are generic parameters, so functions using lifetimes are actually generic functions
- Structs using lifetimes are also generic types. If you want to use a reference in a struct, you need to annotate lifetimes

```
struct Person<'a> {
    first: &'a str,
    last: &'a str
}
```

```rust
// This works thanks to lifetime elision
fn identity(s: &str) -> &str {
    s
}
```

- Each parameter gets its own lifetime (input lifetimes)
- If there is one input lifetime and one output lifetime, the output lifetime gets assigned to the input lifetime
- If there are multiple params, but one of them is &self or &mut self, then the output lifetime gets assigned this lifetime

Quizz

```rust
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Quizz 2: Is there a compilation error

```
fn smallest <'a> (a: &'a str, b: &'a str) -> &'a str {
    if a < b { a } else { b }
}

fn main() {
    let a = "hello";
    let c;
    {
        let b = "world";
        c = smallest(b, a);
        println!("{}", c);
        let d = b;
    }
    println!("{}", c);
}
```

AdaCore

Quizz 3: Is there a compilation error

```rust
#[derive(Debug)]
struct Person<'a> {
    first: &'a str,
    last: &'a str
}

fn main() {
    let first = "Raphael".to_string();
    let p;

    {
        let last = "Amiard".to_string();
        p = Person { first: &first, last: &last };
    }
    println!("{:?}", p);
}
```

Quizz 4: Is there a compilation error

```
#[derive(Debug)]
struct Person<'a> {
    first: &'a str,
    last: &'a str
}

fn main() {
    let first = "Raphael".to_string();
    let p;

    {
        let last = "Amiard".to_string();
        p = Person { first: &first, last: &last };
        println!("{:?}", p);
    }
}
```

AdaCore

Packages & modularity

Modularity (1)

- Rust's compilation model is different from C/C++
- Also very different from Ada
- Rust's compilation unit is the crate
- A crate can span several files, and is usually much bigger than an Ada or C compilation unit (C++ is different because of templates)

Consequence is that parallel compilation is hampered in Rust.

- Rust compiler is incremental on a sub-file level

AdaCore

Modularity (2)

- Two types of crates: Binary crates and library crates
    - Entry point for binary crates: main.rs
    - Entry point for library crates: lib.rs
    - Both can be redefined
- Generally, a library = a crate (but a Cargo package can contain one or more crates)
- A crate can be subdivided in modules

A crate can be further subdivided into modules

- Modules provide scoping, organization, and encapsulation
- A module can be defined:
    - Inline
    - In a file corresponding to the module name
- By default, a module is private
- By default, items in a module are private

```
// Inline module
pub mod ExprEval {
    pub struct Expr {
    }

    ...
}
```

Modules

```rust
// Module in a separate file

// main.rs

pub mod ExprEval

// expreval.rs

pub struct Expr {
}
```

```rust
// Module in a separate file, in a nested dir

// main.rs

pub mod ExprEval

// expreval.rs

pub mod Eval;

pub struct Expr {
}

// expreval/eval.rs

pub fn eval(...)
```

Functional programming

- In Rust, functions and closures are different
- Closures can be nested in functions, and can capture functions from their environment, which regular functions cannot

```
fn main() {
    let y = 12;
    let adder = |x| x + y;
    println!("{}", adder(12));
}
```

- External variables are captured via borrow, so regular borrow rules apply!
- You can explicitly move captured values

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

Functional programming: Closures and iterators

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    let sum = v.iter()
    .map(|el| el * el)
    .reduce(|acc, el| acc + el);

    println!("{}", sum.unwrap());

    v.iter().for_each(|el| {
        println!("{}", el);
    })
}
```

- Rust has *many* methods like this on iterators

```rust
fn main() {
    let v = HashMap::from([
        ("one", 1),
        ("two", 2)
    ]);

    let v2: HashMap<i32, &str> =
        v.iter().map(|(x, y)| (*y, *x)).collect();

    println!("{:?}", v2);
}
```

Quizz 1: Does this compile

```
fn main() {
    let mut y = 12;
    let adder = |x| x + y;
    y = 15
    println!("{}", adder(12));
}
```

```rust
use std::cell::RefCell;

fn main() {
    let y = RefCell::new(12);
    let adder = |x| x + *y.borrow();
    *y.borrow_mut() = 15;
    println!("{}", adder(12));
}
```

AdaCore

Quizz 3: Does this compile

```
use std::cell::RefCell;

struct Adder {
    adder_fn: Box<dyn Fn(i32) -> i32>
}

fn create_adder(val: RefCell<i32>) -> Adder {
    Adder {adder_fn: Box::new(|x| x + *val.borrow())}
}

fn main() {
    let v = RefCell::new(12);
    let adder = create_adder(v);
    println!("{}", *v.borrow());
}
```

AdaCore

Quizz 4: Does this compile

```rust
use std::cell::RefCell;
use std::rc::Rc;

struct Adder {
    adder_fn: Box<dyn Fn(i32) -> i32>
}

fn create_adder(val: Rc<RefCell<i32>>) -> Adder {
    Adder {adder_fn: Box::new(move |x| x + *val.borrow())}
}

fn main() {
    let v = Rc::new(RefCell::new(12));
    let adder = create_adder(v.clone());
    println!("{}", (adder.adder_fn)(12));
    *v.borrow_mut() = 15;
    println!("{}", (adder.adder_fn)(12));
}
```

AdaCore

Error handling

- Rust has no exceptions
- The closest thing it has is unrecoverable errors (via `panic!`)
- Obviously not a solution for robust applications

```
fn main() {
    let v = vec![1, 2, 3];

    v[99]; // PANIC
}
```

Backtraces

When your program panics, running it with *RUST_BACKTRACE=1* will show you a backtrace:

```
$ RUST_BACKTRACE=1 cargo run
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 99', src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/3b348d932aa5c9884310d025cf7c516023fd0d9a/library/std/src/panicking.rs:584:5
   1: core::panicking::panic_fmt
             at /rustc/3b348d932aa5c9884310d025cf7c516023fd0d9a/library/core/src/panicking.rs:143:
   2: core::panicking::panic_bounds_check
             at /rustc/3b348d932aa5c9884310d025cf7c516023fd0d9a/library/core/src/panicking.rs:85:5
   3: <usize as core::slice::index::SliceIndex<[T]>>::index
             at /rustc/3b348d932aa5c9884310d025cf7c516023fd0d9a/library/core/src/slice/index.rs:18
   4: core::slice::index::<impl core::ops::index::Index<I> for [T]>::index
             at /rustc/3b348d932aa5c9884310d025cf7c516023fd0d9a/library/core/src/slice/index.rs:15
   5: <alloc::vec::Vec<T,A> as core::ops::index::Index<I>>::index
             at /rustc/3b348d932aa5c9884310d025cf7c516023fd0d9a/library/alloc/src/vec/mod.rs:2531:
   6: test_epita::main
             at ./src/main.rs:4:5
   7: core::ops::function::FnOnce::call_once
             at /rustc/3b348d932aa5c9884310d025cf7c516023fd0d9a/library/core/src/ops/function.rs:2
```

- Proper way to handle errors is via the *Result<T>* type (shown earlier).
- TIP: Main can return a *Result* (but only with () as an OK type):
- Rust provides the *?* operator for easy(er) error handling

```
use std::num::ParseIntError;

fn main() -> Result<(), ParseIntError> {
    let number_str = "10a";

    let n = number_str.parse::<i32>()?;
    //                              ^ Either unwrap, or return the error result
    println!("{}", n);
    Ok(())
}
```

Result (2)

You can also use early return for easier error handling

```
use std::num::ParseIntError;

fn main() -> Result<(), ParseIntError> {
    let numbers = ["12", "15", "18a"];
    let mut n = 0;

    for num in numbers {
        match num.parse::<i32>() {
            Ok(val) => { n += val; }
            Err(e) => {
                return Err(e);
            }
        }
    };

    Ok(())
}
```

Smart pointer types

Box is a simple reference. Used when you want to *store* a reference, rather than just *borrow* it (see the expression evaluator exercise).

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

- You cannot have multiple references to a box!!

```rust
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

```rust
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```