Assignment 6: RIIR

Due: Wednesday, November 8, 2017 at 4:20pm
Submission deadline: Saturday, November 11, 2017 at 4:20pm

In order to be a trendsetting programming language aficionado, when confronted with a new application or system, you always have to ask the key question: have you considered Rewriting It In Rust? The course staff have decided the answer is yes, and you're going to do it! This assignment will have you take three small applications written in Lua and OCaml and reimplement them in Rust. The learning goals are to understand how our knowledge can transfer from one language to another and get used to the basic Rust workflow.

Setup

In order to do this assignment, you must first setup Rust.

Copying the files is same as always—copy the assignment out of our folder on Rice.

On the Rice machines:

```
cp -r /afs/ir/class/cs242/assignments/assign6 assign6
```

On your local machine:

```
scp -r <SUNetID>@rice.stanford.edu:/afs/ir/class/cs242/assignments/assign6 assign6
```

Requirements

You will turn in these files:

- `part1/rust/src/main.rs`
- `part2/rust/src/lib.rs`
- `part3/rust/src/lib.rs`

There are no written portions to this assignment.

Prelude: Modules and Cargo

Recall in OCaml that modules have two purposes—they function both as a way of *namespacing* code (i.e. associating groups functions with a single name) as well as a way of declaring interfaces. In Rust, traits are used for interfaces, and modules just fulfill the former function (namespacing). If you want to import a module, you can put a `use` statement at the top of your Rust program, e.g.

```
use std::io;
use std::io::Write;

fn main() {
    io::stdout().write("Hi!".as_bytes());
}
```

Functions are accessed from module namespaces using the `::` operator. Additionally, like in OCaml, each file defines a module, although you can declare mini-modules within a file using the `mod` keyword.

In order to build your code, instead of using `rustc` like in lecture, we will use the canonical Rust build tool and package manager, Cargo. Cargo defines a new unit of compilation called a "crate", which is basically a top-level module

plus its dependencies. For this assignment, each part has a standalone crate.

Crates are either binaries (they build some executable you can run, `main.rs`) or libraries (they produce some functions you can link against, `lib.rs`). Part 1 in this assignment is a binary, whereas parts 2 and 3 are libraries. See the testing code at the bottom of each section for the appropriate Cargo command to run.

Part 1: Password Checker (20%)

The first application is a small password checker demonstrating the usage of I/O, strings, and loops. The game is a program which reads lines from standard in, compares them to a secret password, and then either prints "You guessed it!" if the user inputs the secret value or prints the total number of guesses so far otherwise. For example, if our password is "Password1", here's a sample session:

```
test
Guesses: 1
hello!
Guesses: 2
Password1
You guessed it!
```

We have provided you an OCaml implementation at `part1/ocaml/guesser.ml`, and you are to write the corresponding Rust implementation at `part1/rust/src/main.rs`. A few notes:

- In OCaml, `In_channel.input_line In_channel.stdin` reads a line from stdin and discards the trailing newline character.
- In Rust, you can read a line from stdin with `read_line`. You may assume in your implementation that `read_line` will never fail.
- For this and the remaining tasks, consider which parts of the implementation should fundamentally differ between Rust and OCaml. For example, here, your `main` function in Rust should not be recursive—you should use the `loop` construct instead.
- Your output should exactly match ours, as our grading script will do a precise string comparison.

To test your code, go into the `part1/rust` directory and run:

```
cargo run
```

Part 2: Lists and Trees (30%)

In this section, you will implement a number of small functions relating to vectors and trees. For each of the list functions, we have provided you an equivalent or similar function in `part2/lua/list.lua`.

2a: `add_n` (10%)

For starters, to warm up with using vectors, implement the following functions:

```
pub fn add_n(v: Vec<i32>, n: i32) -> Vec<i32>;
pub fn add_n_inplace(v: &mut Vec<i32>, n: i32);
```

Where `add_n(v, n)` returns a new vector with `n` added to each element of the input, and `add_one_inplace(v, n)` performs the same add operation but mutates the input vector in-place. To implement these functions, you can either use a for loop or explore using Iterator::map.

2b: `reverse` (10%)

Our next goal is to write a generic function that takes any kind of vector and reverses it in-place. Here's an incorrect attempt at a solution:

```rust
pub fn reverse_clone<T>(v: &mut Vec<T>) {
    let n = v.len();
    for i in 0..n/2 {
        let x: T = v[i];
        v[i] = v[n-i-1];
        v[n-i-1] = x;
    }
}
```

This solution seems correct, but when we compile it, we get the following error:

```
error[E0507]: cannot move out of indexed content
  --> src/lib.rs:32:20
   |
32 |         let x: T = v[i];
   |                    ^^^^
```

The issue is that when we use the index operator (`[]`), we get back the actual object at the requested index, not a pointer or reference, i.e. we try to move the element out of the vector. However, this would require transferring ownership of the entire vector as well, which we want to avoid. One workaround would be to avoid a move by using a clone instead:

```rust
pub fn reverse_clone<T: Clone>(v: &mut Vec<T>) {
    let n = v.len();
    for i in 0..n/2 {
        let x: T = v[i].clone();
        v[i] = v[n-i-1].clone();
        v[n-i-1] = x;
    }
}
```

> Note: recall that `T: Clone` is a *trait bound* that defines what interface the type `T` must fulfill. Here, `T: Clone` means that "any type `T` that has the `.clone()` method implemented."

This works correctly, but is incredibly inefficient—it requires us to copy every single element in the vector in order to reverse it in-place, so there was no point in being in-place! Your task is to implement the above function *but without using a clone*:

```rust
pub fn reverse<T>(v: &mut Vec<T>);
```

Rust provides no way to do this safely (aside from relevant library functions), so you will need to explore using unsafe methods, specifically the `ptr::swap` function. Do not use `Vec::reverse` or `Vec::swap` (that would defeat the point).

2c: `Tree` (10%)

Will was attempting to define a new datatype for a generic binary tree, similar to the one shown in class:

```rust
pub enum Tree<T> {
    Node(Box<Tree<T>>, T, Box<Tree<T>>),
    Leaf
}
```

This is intended to approximate to the following OCaml datatype:

```
type 'a tree =
  | Node of 'a tree * 'a * 'a tree
  | Leaf
```

To create a tree, Will also defined a "wrap" function that takes a tree and converts it into a pointer that can be used as a branch in another tree.

```rust
fn wrap<T>(t: Tree<T>) -> Box<Tree<T>>;
```

This worked fine and dandy until Will wanted to *reuse* a node as the branch of two parents (i.e. as a directed acyclic graph). For example, Will wrote the following test:

```
#[test]
fn tree_test() {
    let t1 = wrap(Tree::Node(
        wrap(Tree::Leaf),
        100,
        wrap(Tree::Leaf)));
    let t2 = Tree::Node(wrap(Tree::Leaf), 30, t1.clone());
    let t3 = Tree::Node(wrap(t2), 40, t1.clone());

    // Should produce the tree:
    //    40
    //   /  \
    //  30   \
    //   \   /
    //    100
}
```

But oh no! This test failed to compile with the following error:

```
error[E0599]: no method named `clone` found for type `std::boxed::Box<Tree<{integer}>>` in the current scope
  --> src/lib.rs:70:54
   |
70 |         let t2 = Tree::Node(wrap(Tree::Leaf), 30, t1.clone());
   |                                                      ^^^^^
   |
   = note: the method `clone` exists but the following trait bounds were not satisfied:
           `Tree<{integer}> : std::clone::Clone`
```

Your task is to modify the type of `Tree` and the type/implementation of `wrap` to make this test work (without modifying the test!). You should not add any additional trait bounds on `T` —instead think about using one of the smart pointer types we discussed in class. What sort of memory management might be appropriate here?

To test your code, go into the `part2/rust` directory and run:

```
cargo test
```

Note that if you are testing parts 2a and 2b, you will likely want to comment out the `tree_test` until you are working on 2c.

Part 3: Graph Library (50%)

In the final section, you will port a graph library from OCaml to Rust. Specifically, you will implement a graph fulfilling the following interface:

```
pub trait Graph<T> {
    type Node;
    fn add_node(&mut self, x: T) -> Self::Node;
    fn add_edge(/* you will fill this in */) -> /* ?? */;
    fn neighbors(/* you will fill this in */) -> /* ?? */;
    fn value(&self, n: Self::Node) -> &T;
    fn connected(&self, n1: Self::Node, n2: Self::Node) -> bool;
}
```

> Note: the `type Node` is an example of an associated type. Here, writing `Self::Node` means "the `Node` type within the `Graph` trait." It allows us to reference the `Node` type in the trait without having concrete knowledge of what `Node` is.

A corresponding OCaml implementation is provided in `ocaml/graph.ml`. You will implement the `AdjListGraph` structure in `rust/src/lib.rs`, an adjacency list implementation of a directed graph. Here, nodes are integers

(`Node = i32`), and a graph tracks both the mapping from nodes to node values (`HashMap<i32, T>`) and the adjacency list mapping nodes to nodes on outgoing edges (`HashMap<i32, Vec<i32>>`).

Implementing the graph will require an understanding of two new data structures, the `HashSet` and `HashMap`. They function similar to the `Int.Set.t` and `Int.Map.t` in OCaml, except they are not functional data structures and can be updated in-place. Unlike everything else in this course, both data structures actually have good documentation and Google-able FAQs, so refer to the docs on usage.

Look at `ocaml/graph.mli` for documentation on the expected behavior of each function as well as the tests in `lib.rs` for example usage in Rust.

Two hints:

- You can assume that any integers passed to `neighbors` or `value` are valid nodes, i.e. they were created by `add_node(...)`. As a corollary, when using functions like `HashMap::get` you can use the function `Option::unwrap` to assume an option is a Some.
- To convert a vector into a HashSet, you can use the `Iterator::cloned` and `Iterator::collect` functions.

`add_edge` and `neighbors`

In the hustle and bustle of building the new assignments, your busy TAs accidentally left off function definitions for `neighbors` and `add_edge` from the `Graph` trait. Whoops! Luckily, they at least left behind a few tests demonstrating how they're expected to be used.

Your task in addition to implementing the interface is to fill in an interface for the `add_edge` and `neighbors` functions that matches the expected usage provided in the tests. Look at the other functions in the `Graph` trait as well as the OCaml type signatures for guidance. Note that the choice of types will be graded, so use the most concise types for the functions as possible.

To test your code, go into the `part3/rust` directory and run:

```
cargo test
```

Note: this file won't compile until you have defined `add_edge` and `neighbors`

Submitting

Once you have completed the assignment, upload the files in your copy of `assign6` to Rice. To submit, navigate to the assignment root and execute:

```
python /afs/ir/class/cs242/bin/submit.py 6
```