part10.rs

Rust-101, Part 10: Closures

Assume we want to write a function that does *something* on, say, every digit of a `BigInt`. We will then need a way to express the action that we want to be taken, and to pass this to our function. In Rust, a natural first attempt to express this is to have a trait for it.

So, let us define a trait that demands that the type provides some method `do_action` on digits. This immediately raises the question: How do we pass `self` to that function? Owned, shared reference, or mutable reference? The typical strategy to answer this question is to use the strongest type that still works. Certainly, passing `self` in owned form does not work: Then the function would consume `self`, and we could not call it again, on the second digit. So let's go with a mutable reference.

Now we can write a function that takes some `a` of a type `A` such that we can call `do_action` on `a`, passing it every digit.

Remember that the `mut` above is just an annotation to Rust, telling it that we're okay with `a` being mutated. Calling `do_action` on `a` takes a mutable reference, so mutation could indeed happen.

As the next step, we need to come up with some action, and write an appropriate implementation of `Action` for it. So, let's say we want to print every digit, and to make this less boring, we want the digits to be prefixed by some arbitrary string. `do_action` has to know this string, so we store it in `self`.

Here we perform the actual printing of the prefix and the digit. We're not making use of our ability to change `self` here, but we could replace the prefix if we wanted.

Finally, this function takes a `BigInt` and a prefix, and prints the digits with the given prefix. It does so by creating an instance of `PrintWithString`, giving it the prefix, and then passing that to `act_v1`. Since `PrintWithString` implements `Action`, Rust now knows what to do.

Here's a small main function, demonstrating the code above in action. Remember to edit `main.rs` to run it.

Closures

Now, as it turns out, this pattern of describing some form of an action, that can carry in additional data, is very common. In general, this is called a *closure*. Closures take some arguments and produce a result, and they have an *environment* they can use, which corresponds to the type `PrintWithString` (or any other type implementing `Action`). Again we have the choice of passing this environment in owned or borrowed form, so there are three traits for closures in Rust: `Fn`-closures get a shared reference, `FnMut`-closures get a mutable reference, and `FnOnce`-closures consume their environment (and can hence be called only once). The syntax for a closure trait which takes arguments of type `T1`, `T2`, ... and returns something of type `U` is `Fn(T1, T2, ...) -> U`.

This defines `act` very similar to above, but now we demand `A` to be the type of a closure that mutates its borrowed environment, takes a digit, and returns nothing.

We can call closures as if they were functions - but really, what's happening here is translated to essentially what we wrote above, in `act_v1`.

Now that we saw how to write a function that operates on closures, let's see how to write a closure.

The syntax for closures is `|arg1, arg2, ...| code`. Notice that the closure can reference variables like `prefix` that it did not take as argument - variables that happen to be present *outside* of the closure. We say that the closure *captures* variables. Rust will now automatically create a type (like `PrintWithString`) for the environment of the closure with fields for every captured variable, implement the closure trait for this type such that the action performed is given by the code of the closure, and finally it will instantiate the environment type here at the definition site of the closure and fill it appropriately.

You can change `main` to call this function, and you should notice - nothing, no difference in behavior. But we wrote much less boilerplate code!

Remember that we decided to use the `FnMut` trait above? This means our closure could actually mutate its environment. For example, we can use that to count the digits as they are printed.

This time, the environment will contain a field of type `&mut usize`, that will be initialized with a mutable reference of `count`. The closure, since it mutably borrows its environment, is able to access this field and mutate `count` through it. Once `act` returns, the closure is destroyed and `count` is no longer borrowed. Because closures compile down to normal types, all the borrow checking continues to work as usually, and we cannot accidentally leak a closure somewhere that still contains, in its environment, a dead reference.

Fun with iterators and closures

If you are familiar with functional languages, you are probably aware that one can have lots of fun with iterators and closures. Rust provides a whole lot of methods on iterators that allow us to write pretty functional-style list manipulation.

Let's say we want to write a function that increments every entry of a `Vec` by some number, then looks for numbers larger than some threshold, and prints them.

`map` takes a closure that is applied to every element of the iterator. `filter` removes elements from the iterator that do not pass the test given by the closure. Since all these closures compile down to the pattern described above, there is actually no heap allocation going on here. This makes closures very efficient, and it makes optimization fairly trivial: The resulting code will look like you hand-rolled the loop in C.

Sometimes it is useful to know both the position of some element in a list, and its value. That's where the `enumerate` function helps.

`enumerate` turns an iterator over `T` into an iterator over `(usize, T)`, where the first element just counts the position in the iterator. We can do pattern matching right in the loop header to obtain names for both the position, and the value.

And as a final example, one can also collect all elements of an iterator, and put them, e.g., in a vector.

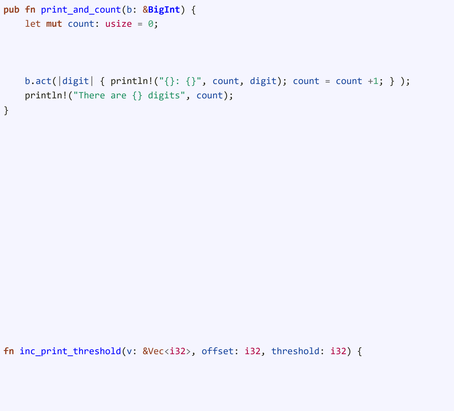
Here, the return type of `collect` is inferred based on the return type of our function. In general, it can return anything implementing `FromIterator`. Notice that `iter` gives us an iterator over borrowed `i32`, but we want to own them for the result, so we insert a `map` to dereference.

Exercise 10.1: Look up the documentation of `Iterator` to learn about more functions that can act on iterators. Try using some of them. What about a function that sums the even numbers of an iterator? Or a function that computes the product of those numbers that sit at odd positions? A function that checks whether a vector contains a certain number? Whether all numbers are smaller than some threshold? Be creative!

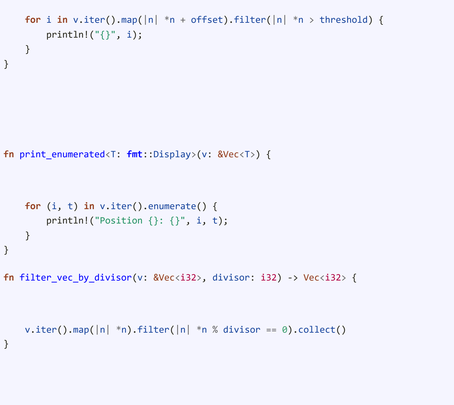
Exercise 10.2: We started the journey in Part 02 with `SomethingOrNothing<T>`, and later learned about `Option<T>` in Part 04. `Option<T>` also has a `map` function. Read its documentation here. Which functions in previous parts can you rewrite to use `map` instead? (Hint: read the source code of `map`, and see if the pattern appears in your own code.) Bonus: `test_invariant` in Part 05 doesn't use `match`, but can you still find a way to rewrite it with `map`?


```
use std::fmt;
use part05::BigInt;

trait Action {
    fn do_action(&mut self, digit: u64);
}

impl BigInt {
    fn act_v1<A: Action>(&self, mut a: A) {

        for digit in self {
            a.do_action(digit);
        }
    }
}
struct PrintWithString {
    prefix: String,
}

impl Action for PrintWithString {

    fn do_action(&mut self, digit: u64) {
        println!("{}{}", self.prefix, digit);
    }
}

fn print_with_prefix_v1(b: &BigInt, prefix: String) {
    let my_action = PrintWithString { prefix: prefix };
    b.act_v1(my_action);
}

pub fn main() {
    let bignum = BigInt::new(1 << 63) + BigInt::new(1 << 16) + BigInt::new(1 << 63);
    print_with_prefix_v1(&bignum, "Digit: ".to_string());
}
```

```
impl BigInt {
    fn act<A: FnMut(u64)>(&self, mut a: A) {
        for digit in self {

            a(digit);
        }
    }
}
pub fn print_with_prefix(b: &BigInt, prefix: String) {

    b.act(|digit| println!("{}{}", prefix, digit) );
}
```



```
pub fn print_and_count(b: &BigInt) {
    let mut count: usize = 0;

    b.act(|digit| { println!("{}: {}", count, digit); count = count +1; } );
    println!("There are {} digits", count);
}
```



```
fn inc_print_threshold(v: &Vec<i32>, offset: i32, threshold: i32) {

    for i in v.iter().map(|n| *n + offset).filter(|n| *n > threshold) {
        println!("{}", i);
    }
}

fn print_enumerated<T: fmt::Display>(v: &Vec<T>) {

    for (i, t) in v.iter().enumerate() {
        println!("Position {}: {}", i, t);
    }
}
fn filter_vec_by_divisor(v: &Vec<i32>, divisor: i32) -> Vec<i32> {

    v.iter().map(|n| *n).filter(|n| *n % divisor == 0).collect()
}
```