part07.rs

Rust-101, Part 07: Operator Overloading, Tests, Formatting

With our new knowledge of lifetimes, we are now able to write down the desired type of `min`: We want the function to take two references *with the same lifetime*, and then return a reference with that lifetime. If the two input lifetimes would be different, we would not know which lifetime to use for the result.

Now we can implement a generic function `vec_min` that works on above trait. The code is pretty much straight-forward, and Rust checks that all the lifetimes actually work out. Observe that we don't have to make any copies!

```
pub use part05::BigInt;
```

```
pub trait Minimum {
    fn min<'a>(&'a self, other: &'a Self) -> &'a Self;
}
```

```
pub fn vec_min<T: Minimum>(v: &Vec<T>) -> Option<&T> {
    let mut min: Option<&T> = None;
    for e in v {
        min = Some(match min {
            None => e,
            Some(n) => n.min(e)
        });
    }
    min
}
```

Notice that the return type `Option<&T>` is technically (leaving the borrowing story aside) a pointer to a `T`, that could optionally be invalid. In other words, it's just like a pointer in C(++) or Java that can be `NULL`! However, thanks to `Option` being an `enum`, we cannot forget to check the pointer for validity, avoiding the safety issues of C(++).
Also, if you are worried about wasting space, notice that Rust knows that `&T` can never be `NULL`, and hence optimizes `Option<&T>` to be no larger than `&T`. The `None` case is represented as `NULL`. This is another great example of a zero-cost abstraction: `Option<&T>` is exactly like a pointer in C(++), if you look at what happens during execution - but it's much safer to use.

Exercise 07.1: For our `vec_min` to be usable with `BigInt`, you will have to provide an implementation of `Minimum`. You should be able to pretty much copy the code you wrote for exercise 06.1. You should *not* make any copies of `BigInt`!

```
impl Minimum for BigInt {
    fn min<'a>(&'a self, other: &'a Self) -> &'a Self {
        unimplemented!()
    }
}
```

Operator Overloading

How can we know that our `min` function actually does what we want it to do? One possibility here is to do *testing*. Rust comes with nice built-in support for both unit tests and integration tests. However, before we go there, we need to have a way of checking whether the results of function calls are correct. In other words, we need to define how to test equality of `BigInt`. Being able to test equality is a property of a type, that - you guessed it - Rust expresses as a trait: `PartialEq`.

Doing this for `BigInt` is fairly easy, thanks to our requirement that there be no trailing zeros. We simply re-use the equality test on vectors, which compares all the elements individually. The `inline` attribute tells Rust that we will typically want this function to be inlined.

```
impl PartialEq for BigInt {
    #[inline]
    fn eq(&self, other: &BigInt) -> bool {
        debug_assert!(self.test_invariant() && other.test_invariant());
        self.data == other.data
    }
}
```

Since implementing `PartialEq` is a fairly mechanical business, you can let Rust automate this by adding the attribute `derive(PartialEq)` to the type definition. In case you wonder about the "partial", I suggest you check out the documentation of `PartialEq` and `Eq`. `Eq` can be automatically derived as well.

Now we can compare `BigInt`s. Rust treats `PartialEq` special in that it is wired to the operator `==`: That operator can now be used on our numbers! Speaking in C++ terms, we just overloaded the `==` operator for `BigInt`. Rust does not have function overloading (i.e., it will not dispatch to different functions depending on the type of the argument). Instead, one typically finds (or defines) a trait that catches the core characteristic common to all the overloads, and writes a single function that's generic in the trait. For example, instead of overloading a function for all the ways a string can be represented, one writes a generic functions over ToString. Usually, there is a trait like this that fits the purpose - and if there is, this has the great advantage that any type *you* write, that can convert to a string, just has to implement that trait to be immediately usable with all the functions out there that generalize over `ToString`. Compare that to C++ or Java, where the only chance to add a new overloading variant is to edit the class of the receiver.

```
fn compare_big_ints() {
    let b1 = BigInt::new(13);
    let b2 = BigInt::new(37);
    println!("b1 == b1: {} ; b1 == b2: {}; b1 != b2: {}", b1 == b1, b1 == b2, b1 != b2);
}
```

Why can we also use `!=`, even though we just overloaded `==`? The answer lies in what's called a *default implementation*. If you check out the documentation of `PartialEq` I linked above, you will see that the trait actually provides two methods: `eq` to test equality, and `ne` to test inequality. As you may have guessed, `!=` is wired to `ne`. The trait *definition* also provides a default implementation of `ne` to be the negation of `eq`. Hence you can just provide `eq`, and `!=` will work fine. Or, if you have a more efficient way of deciding inequality, you can provide `ne` for your type yourself.

Testing

With our equality test written, we are now ready to write our first testcase. It doesn't get much simpler: You just write a function (with no arguments or return value), and give it the `test` attribute. `assert!` is like `debug_assert!`, but does not get compiled away in a release build.

```
#[test]
fn test_min() {
    let b1 = BigInt::new(1);
    let b2 = BigInt::new(42);
    let b3 = BigInt::from_vec(vec![0, 1]);

    assert!(*b1.min(&b2) == b1);
    assert!(*b3.min(&b2) == b2);
}
```

Now run `cargo test` to execute the test. If you implemented `min` correctly, it should all work!

Formatting

There is also a macro `assert_eq!` that's specialized to test for equality, and that prints the two values (left and right) if they differ. To be able to do that, the macro needs to know how to format the value for printing. This means that we - guess what? - have to implement an appropriate trait. Rust knows about two ways of formatting a value: `Display` is for pretty- printing something in a way that users can understand, while `Debug` is meant to show the internal state of data and targeted at the programmer. The latter is what we want for `assert_eq!`, so let's get started.

All formating is handled by `std::fmt`. I won't explain all the details, and refer you to the documentation instead.

In the case of `BigInt`, we'd like to just output our internal `data` array, so we simply call the formating function of `Vec<u64>`.

```
use std::fmt;
```

```
impl fmt::Debug for BigInt {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        self.data.fmt(f)
    }
}
```

`Debug` implementations can be automatically generated using the `derive(Debug)` attribute.

Now we are ready to use `assert_eq!` to test `vec_min`.

```
/*#[test]*/
fn test_vec_min() {
    let b1 = BigInt::new(1);
    let b2 = BigInt::new(42);
    let b3 = BigInt::from_vec(vec![0, 1]);

    let v1 = vec![b2.clone(), b1.clone(), b3.clone()];
    let v2 = vec![b2.clone(), b3.clone()];
    assert_eq!(vec_min(&v1), Some(&b1));
    assert_eq!(vec_min(&v2), Some(&b2));
}
```

Exercise 07.1: Add some more testcases. In particular, make sure you test the behavior of `vec_min` on an empty vector. Also add tests for `BigInt::from_vec` (in particular, removing trailing zeros). Finally, break one of your functions in a subtle way and watch the test fail.

Exercise 07.2: Go back to your good ol' `SomethingOrNothing`, and implement `Display` for it. (This will, of course, need a `Display` bound on `T`.) Then you should be able to use them with `println!` just like you do with numbers, and get rid of the inherent functions to print `SomethingOrNothing<i32>` and `SomethingOrNothing<f32>`.