part08.rs

Rust-101, Part 08: Associated Types, Modules

As our next goal, let us implement addition for our `BigInt`. The main issue here will be dealing with the overflow. First of all, we will have to detect when an overflow happens. This is stored in a so-called *carry* bit, and we have to carry this information on to the next pair of digits we add. The core primitive of addition therefore is to add two digits *and* a carry, and to return the sum digit and the next carry.

So, let us write a function to "add with carry", and give it the appropriate type. Notice Rust's native support for pairs.

Rust's stanza on integer overflows may be a bit surprising: In general, when we write `a + b`, an overflow is considered an *error*. If you compile your program in debug mode, Rust will actually check for that error and panic the program in case of overflows. For performance reasons, no such checks are currently inserted for release builds. The reason for this is that many serious security vulnerabilities have been caused by integer overflows, so just assuming "per default" that they are intended is dangerous.
If you explicitly *do* want an overflow to happen, you can call the `wrapping_add` function (see the documentation, there are similar functions for other arithmetic operations). There are also similar functions `checked_add` etc. to enforce the overflow check.

If an overflow happened, then the sum will be smaller than *both* summands. Without an overflow, of course, it will be at least as large as both of them. So, let's just pick one and check.

The addition did not overflow.
Exercise 08.1: Write the code to handle adding the carry in this case.

Otherwise, the addition *did* overflow. It is impossible for the addition of the carry to overflow again, as we are just adding `0` or `1`.

`overflow_add` is a sufficiently intricate function that a test case is justified. This should also help you to check your solution of the exercise.

Associated Types

Now we are equipped to write the addition function for `BigInt`. As you may have guessed, the `+` operator is tied to a trait (`std::ops::Add`), which we are going to implement for `BigInt`.

In general, addition need not be homogeneous: You could add things of different types, like vectors and points. So when implementing `Add` for a type, one has to specify the type of the other operand. In this case, it will also be `BigInt` (and we could have left it away, since that's the default).

Besides static functions and methods, traits can contain *associated types*: This is a type chosen by every particular implementation of the trait. The methods of the trait can then refer to that type. In the case of addition, it is used to give the type of the result. (Also see the documentation of `Add`.)

In general, you can consider the two `BigInt` given above (in the `impl` line) *input* types of trait search: When `a + b` is invoked with `a` having type `T` and `b` having type `U`, Rust tries to find an implementation of `Add` for `T` where the right-hand type is `U`. The associated types, on the other hand, are *output* types: For every combination of input types, there's a particular result type chosen by the corresponding implementation of `Add`.

Here, we choose the result type to be again `BigInt`.

Now we can write the actual function performing the addition.

We know that the result will be *at least* as long as the longer of the two operands, so we can create a vector with sufficient capacity to avoid expensive reallocations.

Compute next digit and carry. Then, store the digit for the result, and the carry for later. Notice how we can obtain names for the two components of the pair that `overflowing_add` returns.

Exercise 08.2: Handle the final `carry`, and return the sum.

Traits and reference types

If you inspect the addition function above closely, you will notice that it actually consumes ownership of both operands to produce the result. This is, of course, in general not what we want. We'd rather like to be able to add two `&BigInt`.

Writing this out becomes a bit tedious, because trait implementations (unlike functions) require full explicit annotation of lifetimes. Make sure you understand exactly what the following definition says. Notice that we can implement a trait for a reference type!

Exercise 08.3: Implement this function.

Exercise 08.4: Implement the two missing combinations of arguments for `Add`. You should not have to duplicate the implementation.

Modules

As you learned, tests can be written right in the middle of your development in Rust. However, it is considered good style to bundle all tests together. This is particularly useful in cases where you wrote utility functions for the tests, that no other code should use.

Rust calls a bunch of definitions that are grouped together a *module*. You can put the tests in a submodule as follows. The `cfg` attribute controls whether this module is even compiled: If we added some functions that are useful for testing, Rust would not bother compiling them when you just build your program for normal use. Other than that, tests work as usually.

Exercise 08.5: Add some more cases to this test.

As already mentioned, outside of the module, only those items declared public with `pub` may be used. Submodules can access everything defined in their parents. Modules themselves are also hidden from the outside per default, and can be made public with `pub`. When you use an identifier (or, more general, a *path* like `mod1::submod::name`), it is interpreted as being relative to the current module. So, for example, to access `overflowing_add` from within `my_mod`, you would have to give a more explicit path by writing `super::overflowing_add`, which tells Rust to look in the parent module.

You can make names from other modules available locally with `use`. Per default, `use` works globally, so e.g. `use std;` imports the *global* name `std`. By adding `super::` or `self::` to the beginning of the path, you make it relative to the parent or current module, respectively. (You can also explicitly construct an absolute path by starting it with `::`, e.g., `::std::cmp::min`). You can say `pub use path;` to simultaneously *import* names and make them publicly available to others. Finally, you can import all public items of a module at once with `use module::*;`.

Modules can be put into separate files with the syntax `mod name;`. To explain this, let me take a small detour through the Rust compilation process. Cargo starts by invoking `rustc` on the file `src/lib.rs` or `src/main.rs`, depending on whether you compile an application or a library. When `rustc` encounters a `mod name;`, it looks for the files `name.rs` and `name/mod.rs` and goes on compiling there. (It is an error for both of them to exist.) You can think of the contents of the file being embedded at this place. However, only the file where compilation started, and files `name/mod.rs` can load modules from other files. This ensures that the directory structure mirrors the structure of the modules, with `mod.rs`, `lib.rs` and `main.rs` representing a directory or crate itself (similar to, e.g., `__init__.py` in Python).

Exercise 08.6: Write a subtraction function, and testcases for it. Decide for yourself how you want to handle negative results. For example, you may want to return an `Option`, to panic, or to return `0`.


```
use std::{cmp,ops};
use part05::BigInt;

fn overflowing_add(a: u64, b: u64, carry: bool) -> (u64, bool) {
    let sum = a.wrapping_add(b);
    if sum >= a {
        unimplemented!()
    } else {
        (sum + if carry { 1 } else { 0 }, true)
    }
}
/*#[test]*/
fn test_overflowing_add() {
    assert_eq!(overflowing_add(10, 100, false), (110, false));
    assert_eq!(overflowing_add(10, 100, true), (111, false));
    assert_eq!(overflowing_add(1 << 63, 1 << 63, false), (0, true));
    assert_eq!(overflowing_add(1 << 63, 1 << 63, true), (1, true));
    assert_eq!(overflowing_add(1 << 63, (1 << 63) -1 , true), (0, true));
}
impl ops::Add<BigInt> for BigInt {
    type Output = BigInt;
    fn add(self, rhs: BigInt) -> Self::Output {
        let max_len = cmp::max(self.data.len(), rhs.data.len());
        let mut result_vec:Vec<u64> = Vec::with_capacity(max_len);
        let mut carry = false; /* the current carry bit */
        for i in 0..max_len {
            let lhs_val = if i < self.data.len() { self.data[i] } else { 0 };
            let rhs_val = if i < rhs.data.len() { rhs.data[i] } else { 0 };
            let (sum, new_carry) = overflowing_add(lhs_val, rhs_val, carry);
            result_vec.push(sum);
            carry = new_carry;
        }
        unimplemented!()
    }
}

impl<'a, 'b> ops::Add<&'a BigInt> for &'b BigInt {
    type Output = BigInt;
    fn add(self, rhs: &'a BigInt) -> Self::Output {
        unimplemented!()
    }
}

#[cfg(test)]
mod tests {
    use part05::BigInt;

    /*#[test]*/
    fn test_add() {
        let b1 = BigInt::new(1 << 32);
        let b2 = BigInt::from_vec(vec![0, 1]);
        assert_eq!(&b1 + &b2, BigInt::from_vec(vec![1 << 32, 1]));
    }
}
```