Generic types, traits and lifetimes

30 October 2026

Instructions

Your task is to implement types representing simple programs in a simple programming language. In the end, using the interface we define will look like this:

```
let mut program = seq(
    print(when(constant("y"), 1, 2)),
    print(when(constant("x"), 1, 2)),
);

let context = HashMap::from([("x", 0), ("y", 10)]);
program.exec_stmt(&context);
```

`when` returns its second or third argument depending on the value of the first one. The call at the end executes the whole program stored in `program`. In this case, the console output will be

```
1
2
```

As you write the code, add to `main` any uses of the structures you implement.

1. Download the starter code file.
2. Implement the functionality described in the next section.
3. Fix all compiler warnings.
4. Check that the program passes the tests (`cargo test`).
5. Find common problems in the code using `cargo clippy` and fix them.

Functionality

1. Add a trait `Expr` (*expression*) providing one method:
   `fn exec_expr(&mut self, context: &Context) -> u64`.
   It will be the trait of types representing elements of the language that return something. `Context` is an alias for `HashMap<&'static str, u64>`. It will be a map containing the values of constants in a given execution of the program.
2. Add a trait `Stmt` (*statement*) providing one method:
   `fn exec_stmt(&mut self, context: &Context)`.
   It will be the trait of types representing elements of the language that return nothing.
3. Define a struct `Print<T: Expr>` with one field: `inner: T`.
4. Define a generic function `print<T: Expr>(inner: T) -> Print<T>` that wraps the given argument in `Print`.

5. Implement the `Stmt` trait for `Print`. Executing it first executes `inner`, and then prints its value on the console. This pattern of three definitions—a struct, a constructor function and a trait implementation—will repeat for every element of our language. In the following points, unless stated otherwise, "Implement `X` as `Y`" means: implement the struct `X`, the function `x` and the trait `Y` for `X`.
6. Implement `Nothing` as `Stmt`. `Nothing` does nothing when executed and has no fields.
7. Implement `Seq<T: Stmt, U: Stmt>` as `Stmt`. `Seq` has two fields: `first: T` and `second: U`. When executed, it executes the first one and then the second one.
8. Implement three conditionally implemented methods for `Seq`:
   (a) `shorten_1(self) -> T`, if it has the form `Seq<T, Nothing>`. This method returns the first field of the struct.
   (b) `shorten_2(self) -> T`, if it has the form `Seq<Nothing, T>`. This method returns the second field of the struct.
   (c) `collapse(self) -> Nothing`, if it has the form `Seq<Nothing, Nothing>`. This method returns `Nothing`.
9. Implement the `Expr` trait for `u64`. When executed, a `u64` returns its value.
10. Implement `When<?>` as `Expr`. `When` has three fields:
    `condition: ?`, `if_true: ?`, `if_false: ?`. It works like `if`, but `if` is a keyword, so we cannot use it. When executed, `condition` is executed first. If its value is `0`, we execute `if_false` and return its value. Otherwise we do the same with `if_true`. Replace `?` in this point yourself, so that the behaviour matches the description and the tests.
11. Implement `Repeat<const N: u32, T: Stmt>` as `Stmt`. `Repeat` has a field `inner: T` and executes it `N` times.
12. Implement `Constant` as `Expr`. `Constant` has one field, `name: &'static str`, and when executed returns the element of `context` under the key `name`.
13. Implement `ReadFrom<'a>` as `Expr`. `ReadFrom` has one field, `&'a u64`, which is a reference to any variable. When executed, it returns the value of that variable.
14. Implement `SaveIn<?>` as `Expr`. `SaveIn` has two fields:
    `destination: ?` and `inner: ?`. `destination` is a mutable reference to a variable, and `inner` is an expression. When executed, `SaveIn` executes `inner`, saves the value in `destination` and returns it. Replace `?` in this point yourself, so that the behaviour matches the description and the tests.
15. Implement `Volatile<?>` as `Expr`. `Volatile` works like `SaveIn`, but has three fields: `destination: ?`, `name: &'? str` and `inner: ?`. Additionally, before executing `inner`, it creates a new context based on the current one and replaces the value under the key `name` in it with the contents of `destination`. This way the program can change the value in `destination` based on the value currently stored there.

Requirements

- The code compiles with the stable Rust compiler.
- No warnings from the compiler or `clippy`, including no warnings about unused structs or methods.
- All tests pass.
- Full implementation of the specified functionality.

Grading (3 pts)

- 1 pt: Work during the laboratory.
- 1 pt: Full functionality (complete implementation meeting the requirements).
- 1 pt: Presentation of the solution and answers to the instructor's questions.