Typy generyczne, cechy i okresy życia

31 października 2025

Instrukcja

Twoim zadaniem jest implementacja typów reprezentujących proste programy prostego języka programowania. Używanie zdefiniowanego przez nas interfejsu będzie wyglądać na końcu tak:

```
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` zwraca drugi lub trzeci argument w zależności od wartości pierwszego. Wywołanie na końcu spowoduje wykonanie całego programu zapisanego w `program`. W tym wypadku na konsolę zostanie wypisane

```
1
2
```

W miarę pisania kodu dodawaj do `main` dowolne użycia kolejnych implementowanych przez siebie struktur.

1. Pobierz plik z kodem startowym.
2. Zaimplementuj funkcjonalność opisaną w kolejnej sekcji.
3. Napraw wszystkie ostrzeżenia kompilatora.
4. Sprawdź, czy program przechodzi testy (`cargo test`).
5. Znajdź powszechne problemy z kodem używając `cargo clippy` i je napraw.

Funkcjonalność

1. Dodaj cechę `Expr` (*expression*—wyrażenie) udostępniającą jedną metodę:
   `fn exec_expr(&mut self, context: &Context) -> u64`.
   Będzie to cecha typów reprezentujących elementy języka, które coś zwracają. `Context` jest aliasem dla `HashMap<&'static str, u64>`. Będzie to mapa zawierająca wartości stałych w danym wykonaniu programu.
2. Dodaj cechę `Stmt` (*statement*—stwierdzenie) udostępniającą jedną metodę:
   `fn exec_stmt(&mut self, context: &Context)`.
   Będzie to cecha typów reprezentujących elementy języka, które niczego nie zwracają.
3. Zdefiniuj strukturę `Print<T: Expr>`, która zawiera jedno pole: `inner: T`.

4. Zdefiniuj funkcję generyczną `print<T: Expr>(inner: T) -> Print<T>`, która dany argument zawija w `Print`.

5. Zaimplementuj cechę `Stmt` dla `Print`. Wywołanie działa tak, że najpierw wywołany zostaje `inner`, a następnie jego wartość wyświetlona jest na konsolę. Taki schemat trzech definicji: struktura, funkcja konstruująca i implementacja cechy będzie się powtarzać dla każdego elementu naszego języka. W poniższych punktach o ile nie jest napisane inaczej, to tak należy traktować „Zaimplementuj `X` jako `Y`": zaimplementować strukturę `X`, funkcję `x` i cechę `Y` dla `X`.

6. Zaimplementuj `Nothing` jako `Stmt`. `Nothing` nic nie robi przy wywołaniu i nie ma składowych.

7. Zaimplementuj `Seq<T: Stmt, U: Stmt>` jako `Stmt`. `Seq` ma dwa pola: `first: T` i `second: U`. Przy wywołaniu najpierw wywołuje pierwsze, a potem drugie.

8. Zaimplementuj dla `Seq` trzy metody implementowane warunkowo:

   (a) `shorten_1(self) -> T`, jeśli mamy postać `Seq<T, Nothing>`. Metoda ta zwraca pierwsze pole struktury.

   (b) `shorten_2(self) -> T`, jeśli mamy postać `Seq<Nothing, T>`. Metoda ta zwraca drugie pole struktury.

   (c) `collapse(self) -> Nothing`, jeśli mamy postać `Seq<Nothing, Nothing>`. Metoda ta zwraca `Nothing`.

9. Zaimplementuj cechę `Expr` dla `u64`. `u64` przy wykonaniu zwraca swoją wartość.

10. Zaimplementuj `When<?>` jako `Expr`. `When` ma trzy pola:
    `condition: ?`, `if_true: ?`, `if_false: ?`. Działa podobnie do `if`, ale `if` jest słowem kluczowym, więc nie możemy go użyć. Przy wykonaniu najpierw wykonywane jest `condition`. Jeśli jego wartość to `0`, to wykonujemy i zwracamy wartość z `if_false`. W przeciwnym razie robimy to samo z `if_true`. `?` w tym podpunkcie zastąp samodzielnie, aby działanie mogło być zgodne z opisem i testami.

11. Zaimplementuj `Repeat<const N: u32, T: Stmt>` jako `Stmt`. `Repeat` ma pole `inner: T` i wykonuje je `N` razy.

12. Zaimplementuj `Constant` jako `Expr`. `Constant` ma jedno pole, `name: &'static str`, i przy wywołaniu zwraca z `context` element pod kluczem `name`.

13. Zaimplementuj `ReadFrom<'a>` jako `Expr`. `ReadFrom` ma jedno pole, `&'a u64`, które jest referencją na dowolną zmienną. Przy wywołaniu zwraca wartość tej zmiennej.

14. Zaimplementuj `SaveIn<?>` jako `Expr`. `SaveIn` ma dwa pola:
    `destination: ?` i `inner: ?`. `destination` jest referencją mutowalną na zmienną, a `inner` wyrażeniem. Przy wykonaniu `SaveIn` wykonuje `inner`, zapisuje wartość do `destination` i ją zwraca. `?` w tym podpunkcie zastąp samodzielnie, aby działanie mogło być zgodne z opisem i testami.

15. Zaimplementuj `Volatile<?>` jako `Expr`. `Volatile` działa podobnie jak `SaveIn`, ale ma trzy pola: `destination: ? name: &'? str` i `inner: ?`. Dodatkowo, przed wywołaniem `inner` tworzy nowy kontekst na podstawie dotychczasowego, i podmienia w nim wartość pod kluczem `name` na zawartość `destination`. W ten sposób program może zmienić wartość w `destination` na podstawie obecnie tam przechowywanej wartości.

Wymagania

- Kod kompiluje się przy użyciu stabilnego kompilatora Rust.
- Brak ostrzeżeń z kompilatora i `clippy`, w tym brak ostrzeżeń o nieużywanych strukturach lub metodach.
- Wszystkie testy przechodzą.
- Pełna implementacja zadanej funkcjonalności.

Ocena (3 pkt)

- 1 pkt: Praca w trakcie laboratorium.
- 1 pkt: Pełna funkcjonalność (pełna implementacja zgodna z wymaganiami).
- 1 pkt: Prezentacja rozwiązania i odpowiedź na pytania prowadzącego.