Atividade prática 2

Raoni Silva

30 de novembro de 2025

Turma de Estruturas Algébricas 25T56
Unidade 2

Sumário

1 Implementação de `Mimetize` 3
1.1 Relação com categorias 4

2 Mimetizar a random() 4
2.1 Utilizando uma seed 5

3 Mimetizando funções com side-effects 5

4 Void-Bool Cat 6

1 Implementação de `Mimetize`

A implementação foi feita na linguagem de programação `Rust`, utilizando uma classe que guarda um hashmap de entradas e saídas e uma função. No seguinte formato:

Listing 1: Definição de HashMimetize

```rust
pub struct Mimetize<I: Hashable, O, F: Fn(I) -> O> {
    args: HashMap<I, O>,
    f: Box<F>,
}
```

Para atender a requisição de chamar a função apenas uma vez, para utilizar a função interna de Mimetize, é necessário utilizar o método `.call()`. Com isso, é possível adicionar a lógica de "salvar o resultado"da função interna antes de chamar ela de fato.

Listing 2: Como chamar a função de HashMimetize

```rust
fn call(&mut self, i: I) -> O {
    // Busca o resultado no HashMap.
    match self.args.get(&i) {
        // Caso não tenha o resultado, chama f com o
           argumento, e insere no HashMap.
        None => {
            let res = (self.f)(i.clone());
            self.args.insert(i, res.clone());
            return res;
        }
        // Caso já tenha o resultado salvo, apenas retorna
           ele.
        Some(o) => o.clone(),
    }
}
```

A partir desse resultado, é possível utilizar a `HashMimetize` da seguinte forma:

Listing 3: Como utilizar a HashMimetize

```rust
fn call(&mut self, i: I) -> O {
    // Função mimetizada
    let f = |a: String| {
        sleep(Duration::from_secs(2));
        a.len()
    };
    // Mimetize
    let mut m = HashMimetize::new(f);

    // Demora 2s
    let x = m.call(String::from("Argumento"));
    // Instantâneo
    let y = m.call(String::from("Argumento"));
}
```

1.1 Relação com categorias

A partir dessa implementação, o que se pode observar sobre a `HashMimetize`, é que ela mantém a assinatura de $f : I- > O$. E além disso, ela mantém as mesmas entradas e saídas, apesar de ter implementações diferentes. Logo, temos que f e mimetize(f) são o mesmo morfismo na categoria dos tipos.

$$mimetize(f) = f$$

2 Mimetizar a random()

O segundo tópico pedido foi para mimetizar a função $random : Unit- > T$ padrão da linguagem. O jeito de implementar isso é o seguinte:

Listing 4: Mimetizar a random()

```
fn call(&mut self, i: I) -> O {
    let random = |()| {
        let mut r = rng();
        r.random::<i32>()
    };

    let mut m2 = HashMimetize::new(random);

    // Assert falha
    assert_eq!(m2.call(()), random(()));
}
```

Isso demonstra que a mimetização não mantém a igualdade entre morfismos para funções aleatórias.

Do ponto de vista da teoria das categorias, a falha ocorre devido a uma incompatibilidade na definição do domínio do morfismo. A função `mimetize` assume que o argumento f é um morfismo puro, onde para um dado objeto de entrada A, existe um único objeto de saída B. Ou seja, ela enxerga a função como:

$$f : Unit \to T$$

No entanto, a função `random()` depende implicitamente de um estado externo (a *seed* ou o estado do gerador). Para modelar corretamente esse comportamento em uma categoria de funções puras, precisamos explicitar esse estado no domínio e no codomínio. O morfismo real seria algo como:

$$f_{real} : Unit \times World \to T \times World$$

Onde $World$ representa o estado do universo (ou do gerador de números aleatórios) no momento da execução.

Como a função `mimetize` ignora o objeto $World$ e fixa o resultado baseada apenas na entrada 1 (que é constante), ela "congela"o estado do mundo no momento da primeira execução (w_0).

2.1 Utilizando uma seed

Ao utilizar uma seed, é como se utilizassemos o objeto $World$ como entrada, garantindo que a saída será consistente. Traduzindo para o código, é equivalente a:

Listing 5: Mimetizar a random() corretamente

```rust
fn call(&mut self, i: I) -> O {
    let random = |seed: [u8; 32]| {
        let mut r = StdRng::from_seed(seed);
        r.random::<i32>()
    };

    let mut m2 = HashMimetize::new(random);

    // Assert passa
    assert_eq!(m2.call(seed), random(seed));
}
```

Observando sob o ponto de vista das categorias, estamos sendo honestos no nosso morfismo agora, definindo ele como $Seed \to T$.

3 Mimetizando funções com side-effects

Essas duas funções, em rust, foram declaradas da seguinte forma:

Listing 6: Funções com side-effects

```rust
let f1 = |()| {
    println!("Hello!");
    true
};

let f2 = |x: i32| {
    static mut Y: i32 = 0;

    unsafe {
        Y += x;
        return Y;
    }
};
```

Quando a primeira foi mimetizada, observou-se que mesmo chamando ela várias vezes, o side-effect só é ativado uma vez, pois só tem um jeito de chamar a função.

No caso da segunda função, o side-effect é chamado sempre que uma entrada diferente ocorre, ou seja, se for executado f2(5) N vezes, o resultado será o mesmo. Mas caso seja chamado f2(1), f2(2)... O valor na saída sempre vai mudar. Isso ocorre de forma similar a função random(), pois categoricamente, é como se essas funções estivessem ocultando parte da assinatura ($f1 : (Unit \times World \to Bool \times World)$, exibindo apenas parte de seu funcionamento.

4 Void-Bool Cat

Como pedido, a categoria definida pelos objetos Void e Bool e seus morfismos.