Atividade prática 3

Raoni Silva

14 de dezembro de 2025

Turma de Estruturas Algébricas 25T56
Unidade 3

Sumário

1 Estruturas Fundamentais e Morfismos

Esta seção aborda a definição programática de par ordenado (produto) e união disjunta (coproduto), bem como os morfismos básicos entre essas estruturas, para responder a questão 1 da atividade proposta.

1.1 Definição e morfismo de produto

Para definir o produto em teoria das categorias, é necessário definir, para quaisquer objetos A, B de uma categoria, seguinte tripla:

$$(A \times B, \pi_A, \pi_B)$$

Onde π_A e π_B tem a seguinte assinatura:

$$\pi_A : A \times B \to A$$
$$\pi_B : A \times B \to B$$

Dessa forma, é necessário definir o objeto que será o produto, junto de suas projeções esquerdas e direitas. Em Rust, a implementação foi a seguinte:

Listing 1: Definição de produto

```
pub struct Pair<A, B> {
    first: A,
    second: B,
}

impl<A: Clone, B: Clone> Pair<A, B> {
    pub fn left(&self) -> A {
        self.left.clone()
    }

    pub fn right(&self) -> B {
        self.right.clone()
    }
}
```

Onde, *Pair* é o objeto produto ($A \times B$), e os métodos *left* e *right* são as projeções π_A e π_B.

A partir disso, podemos inferir que o morfismo entre dois pares haverá a seguinte assinatura:

$$m : A \times B \to C \times D$$

Como existem diversos possíveis morfismos, eu criei uma interface que define um morfismo entre pares, da forma mais natural:

Listing 2: Morfismo de produto

```
pub trait PairMorfism<A, B, C, D> {
    fn apply(a: Pair<A, B>) -> Pair<C, D>;
}
```

Sob essa modelagem, a definição de morfismos concretos é realizada através de tipos unitários que implementam a interface (trait). Um exemplo clássico é o morfismo de *swap* (ou reversão), que transforma um produto A×B no produto comutado B×A.

Listing 3: Morfismo revert

```
pub struct Revert;

impl<A: Clone, B: Clone> PairMorfism<A, B, B, A> for Revert {
    fn apply(a: Pair<A, B>) -> Pair<B, A> {
        let r = a.right;
        let l = a.left;

        Pair::new(r, l)
    }
}
```

1.2 Definição e morfismo de coproduto

De forma dual ao produto, o coproduto é uma tripla que guarda 3 outras coisas:

$$(A + B, \iota_A, \iota_B)$$

Onde,

$$\iota_A : A \to A + B$$

$$\iota_B : B \to A + B$$

Ou seja, agora em a partir de A e de B eu tenho que conseguir construir/injetar o coproduto A + B, para quem é familiar com java, é como se tivessemos dois construtores, um que recebe um objeto do tipo A, e outro que recebe um objeto do tipo B.

Em termos de implementação, a estrutura que melhor representa uma união disjunta é uma enumeração (Sum Type). Definimos o tipo *Sum* e seus construtores, que atuam como as injeções.

Listing 4: Definição de co-produto

```
pub enum Sum<A, B> {
    Left(A),
    Right(B),
}

impl<A, B> Sum<A, B> {
    pub fn left(a: A) -> Self {
        Self::Left(a)
    }

    pub fn right(b: B) -> Self {
```

```
12          Self::Right(b)
13      }
14  }
```

Onde o *enum Sum* representa o objeto $A + B$, e os construtores estáticos *left* e *right* representam as injeções ι_A e ι_B, respectivamente.

Seguindo a mesma lógica dos produtos, o morfismo entre dois co-produtos terá a seguinte assinatura:

$$m : A + B \to C + D$$

Para representar essa transformação, definimos uma interface de morfismo de soma. Diferente do produto, onde acessamos as projeções, no coproduto precisamos tratar as possibilidades de entrada usando *pattern matching*.

Listing 5: Morfismo de coproduto

```
1  pub trait SumMorfism<A, B, C, D> {
2      fn apply(e: Sum<A, B>) -> Sum<C, D>;
3  }
```

Como exemplo concreto de morfismo, implementamos o *Swap* (comutatividade), que é o dual do morfismo *Revert* apresentado anteriormente. Ele transforma um co-produto $A + B$ em $B + A$, trocando a "direção"da injeção: se o valor veio de A (Left), ele é injetado em B (Right) no destino, e vice-versa.

Listing 6: Morfismo Swap (Comutatividade)

```
1  pub struct Swap;
2
3  impl<A, B> SumMorfism<A, B, B, A> for Swap {
4      fn apply(s: Sum<A, B>) -> Sum<B, A> {
5          match s {
6              Sum::Left(l) => Sum::Right(l),
7              Sum::Right(r) => Sum::Left(r),
8          }
9      }
10 }
```

2 Derivação de produtos

Para essa parte, fiquei pensando bastante em como implementar isso da melhor forma, a partir do enunciado da questão, deu-se a entender que no fim das contas, eu preciso criar uma função que dado um objeto O, ele retorna um produto a partir do O. A assinatura seria algo nesse sentido:

$$deriveProduct : O \to A \times B$$

Lembrando que A e B podem ser também, produtos por si só. A partir dessa lógica, imagino que criar uma função que recebe qualquer objeto que saiba "se decompor em produtos"seja uma necessidade, ou seja, criar uma função que recebe qualquer cara que implemente uma interface *DeriveProduct*. Com isso, como ela

sabe se decompor, eu posso retornar uma tupla a partir de qualquer um objeto desses. A nível de código, segue o contrato definido pela interface:

Listing 7: Derivação do produto

```
pub trait DeriveProduct {
    type Left;
    type Right;

    fn derive(s: Self) -> (Self::Left, Self::Right);
}
```

Perceba que a interface tem seus tipos associados que definem o tipo de sua projeção esquerda e direita, assegurando o tipo do retorno do método *derive*. Segue um exemplo de implementação não trivial para o tipo *User*, que tem três campos, já pra exemplificar a extensão do produto para as triplas, sendo facilmente extensível para qualquer ênuplo.

Listing 8: Exemplo de derivação

```
struct Id(i32);
struct Email(String);
struct Name(String);

struct User {
    id: Id,
    name: Name,
    email: Email,
}

impl DeriveProduct for User {
    type Left = (Id, Name);
    type Right = Email;
    fn derive(s: Self) -> (Self::Left, Self::Right) {
        ((s.id, s.name), s.email)
    }
}
```

Nessa implementação a função *derive* é de fato a implementação da função que deriva produtos automaticamente, para cada tipo que a implementa.

Para utilizar a função derive, basta chamar o método associado ao tipo da seguinte forma:

Listing 9: Utilização da derivação

```
let u = User {
    id: Id(10),
    name: Name("Valdisgleis".to_string()),
    email: Email("valdisgleis@gmail.com".to_string()),
};

let ((id, nome), email) = User::derive(u);
```

Perceba que aqui, devido a implementação de *DeriveProduct* para User retorna uma tupla, onde a projeção esquerda também é uma tupla, nós conseguimos

guardar todas as informações do usuário, decomposto em "produtos puros", que definimos.

3 Derivação de Coprodutos

De forma dual à derivação de produtos, a derivação de coprodutos é definida por meio de uma interface (trait). Embora a estrutura dos tipos associados permaneça similar, a assinatura da função `derive` reflete a natureza de injeção da soma: ela consome o próprio objeto e retorna uma estrutura de soma genérica (`Sum`).

Listing 10: Interface para derivação de coproduto

```
pub trait DeriveSum {
    type Left;
    type Right;

    fn derive(s: Self) -> Sum<Self::Left, Self::Right>;
}
```

Para atender ao requisito de derivar coprodutos a partir de N objetos, utilizamos a propriedade associativa da soma. Computacionalmente, representamos isso através do aninhamento de tipos `Sum`. Dessa forma, a função `derive` é responsável por transformar uma estrutura de escolha com N variantes em uma estrutura "canônica"de somas binárias aninhadas (uma árvore de tipos).

No exemplo a seguir, o `enum` do Rust atua como o recipiente natural para esses N objetos. A implementação da trait mapeia cada variante concreta desse enum para a posição correspondente na estrutura genérica de coproduto.

Listing 11: Implementação para 3 objetos (N=3)

```
struct Id(i32);
struct Email(String);
struct Name(String);

enum UserIdentifier {
    Id(Id),
    Name(Name),
    Email(Email),
}

impl DeriveSum for UserIdentifier {
    type Left = Sum<Id, Name>;
    type Right = Email;

    fn derive(s: Self) -> Sum<Self::Left, Self::Right> {
        match s {
            Self::Id(id) => Sum::Left(Sum::Left(id)),
            Self::Name(name) => Sum::Left(Sum::Right(name)),
            Self::Email(email) => Sum::Right(email),
        }
    }
```

```
}
```

A utilização se dá pela chamada direta da função, que converte o tipo concreto na sua representação categórica genérica:

Listing 12: Implementando o DeriveSum

```
let email = Email("valdisgleis@gmail.com".to_string());
let user = UserIdentifier::Email(email);

let x = UserIdentifier::derive(user);

assert_eq!(x, Sum::Right(email));
```

4 Morfismos entre Produtos

A questão 4 pede, em suma, uma função com a seguinte assinatura:

$$traducao : (t_1 : (A \times B), t_2 : (C \times D), \mathrm{m} : (A \times B \to C \times D)) \to C \times D$$

O objetivo da função é atuar como uma validação computacional do diagrama do produto. Ela aplica o morfismo m em t_1 e verifica se o resultado é consistente com t_2 ao analisar suas projeções. Se as projeções resultantes de $m(t_1)$ forem idênticas às projeções de t_2, a função retorna o par validado; caso contrário, retorna `None`, indicando uma falha na comutatividade ou na expectativa do testemunho.

A implementação em Rust reflete essa lógica, utilizando o tipo `Option` para representar a possibilidade de falha na validação. A assinatura da função foi simplificada para melhorar o entendimento:

Listing 13: Função de tradução/teste

```
fn traducao(t1: (A, B), t2: (C, D), f: F) -> Option<(C, D)>
where
    F: Fn((A, B)) -> (C, D),
{
    let (c2, d2) = f(t1);

    if c2 != t2.0 {
        return None;
    }

    if d2 != t2.1 {
        return None;
    }

    Some((c2, d2))
}
```

5 Gerenciamento de Efeitos Colaterais

A Questão 5 propõe um cenário onde a transformação na primeira coordenada envolve uma função *random*. Na Teoria das Categorias, os morfismos devem satisfazer a transparência referencial: para uma mesma entrada, deve-se obter sempre a mesma saída.

Para utilizar a biblioteca padrão `rand` do Rust sem violar esse princípio, não podemos utilizar o gerador global (que depende de entropia do sistema operacional ou relógio). Devemos utilizar um Gerador de Números Pseudo-Aleatórios (PRNG) deterministicamente inicializado com o valor da entrada.

Abaixo, utilizamos o `StdRng` e a trait `SeedableRng`. Ao semear o gerador com o valor da coordenada inteira do par, garantimos que a "aleatoriedade"seja uma função pura do argumento.

Listing 14: Morfismo com rand determinístico

```
use rand::{Rng, SeedableRng, rngs::StdRng};

use super::*;

fn pure_random(seed: i32) -> i32 {
    let mut rng = StdRng::seed_from_u64(seed as u64);

    rng.random()
}

fn identidade(c: char) -> char {
    c
}

#[test]
fn teste_morfismo_com_seed() {
    let t1: (i32, char) = (42, 'z');

    let morfismo = |(n, c): (i32, char)| -> (i32, char) { (
       pure_random(n), identidade(c)) };

    let valor_esperado = pure_random(42);
    let t2: (i32, char) = (valor_esperado, 'z');

    let resultado = traducao(t1, t2, morfismo);

    assert_eq!(resultado, Some(t2));
    assert_eq!(morfismo((42, 'z')), t2);
}
```