Types algébriques et génériques

Frédéric Wagner

2024

Ensimag

Plan

Types algébriques

Types génériques

Structures génériques classiques

Types sommes

- on a souvent besoin de combiner plusieurs types
  - soit d'un type 1
  - soit d'un type 2
  - ...

Types sommes

- on a souvent besoin de combiner plusieurs types
  - soit d'un type 1
  - soit d'un type 2
  - ...
- type somme

Types sommes

- on a souvent besoin de combiner plusieurs types
  - soit d'un type 1
  - soit d'un type 2
  - ...
- type somme
- exemple
  - primitives svg
    - cercle : cx, cy, r
    - segment : x1, y1, x2, y2

Types sommes

- en rust : **enum**

Types sommes

- en rust : **enum**

```
enum PrimitiveSvg {
  Cercle { rayon: f32, centre: [f32; 2] },
  Segment([[f32; 2]; 2]),
}
let c = PrimitiveSvg::Cercle {
    rayon: 2.4,
    centre: [1.2, 3.5],
};
let s = PrimitiveSvg::Segment([[1.4, 2.3], [2.4, 3.5]]);
```

Autre exemple

- un entier ou rien

```
enum EntierOuRien {
  Entier(u32),
  Rien
}
let e = EntierOuRien::Entier(3);
let r = EntierOuRien::Rien;
```

Utilisation

- nécessite un test
- pattern matching
  - if let / while let
  - match

```
let e = EntierOuRien::Entier(3);
match e {
  EntierOuRien::Entier(e) => println!("{e}"),
  EntierOuRien::Rien => println!("rien"),
}
```

match

- le pattern matching est très sympa

```
let e = EntierOuRien::Entier(3);
let r = EntierOuRien::Rien;
let couple = (e, r);
match couple {
  (EntierOuRien::Entier(_), EntierOuRien::Rien) => (),
  (EntierOuRien::Rien, EntierOuRien::Entier(_)) => (),
  _ => println!("le reste des cas"),
}
```

match

- le pattern matching est très sympa

```
let e = EntierOuRien::Entier(3);
let r = EntierOuRien::Rien;
let couple = (e, r);
match couple {
  (EntierOuRien::Entier(_), EntierOuRien::Rien) => (),
  (EntierOuRien::Rien, EntierOuRien::Entier(_)) => (),
  _ => println!("le reste des cas"),
}
```

NP-difficile

compiler du rust est NP-difficile à cause du match

if let

```
let e = EntierOuRien::Entier(3);
if let EntierOuRien::Entier(vrai_entier) = e {
  assert_eq!(vrai_entier, 3);
  println!("e contient {vrai_entier}");
}
let r = EntierOuRien::Rien;
if let EntierOuRien::Rien = r {
  println!("r est vide");
}
let e = EntierOuRien::Entier(4);
if let EntierOuRien::Entier(vrai_entier) = &e {
  assert_eq!(vrai_entier, &4);
}
```

Plan

Types algébriques

Types génériques

Structures génériques classiques

Types génériques

- paramétrage par une liste de types génériques
  - pour la déclaration de la structure
  - pour les déclarations des méthodes
- un seul code pour plusieurs applications
- une programmation plus abstraite

```
struct Paire<T> {
    premier: T,
    second: T
}
impl<T> Paire<T> {
 fn change_premier(
     &mut self, p: T) {
   self.premier = p
 }
}
```

Types énumérés génériques

- même principe pour les types énumérés

```
enum Prix<E, F> {
    Entier(E),
    Flottant(F)
}
let p:Prix<u32, f64> = Prix::Entier(3u32);
```

Types énumérés génériques

- même principe pour les types énumérés

```
enum Prix<E, F> {
    Entier(E),
    Flottant(F)
}
let p:Prix<u32, f64> = Prix::Entier(3u32);
```

- pourquoi le f64 ici ?

Types énumérés génériques

- même principe pour les types énumérés

```
enum Prix<E, F> {
    Entier(E),
    Flottant(F)
}
let p:Prix<u32, f64> = Prix::Entier(3u32);
```

- pourquoi le f64 ici ?
  - toute variable a toujours un type clairement défini
  - l'inférence de type ne peut pas deviner le type flottant de p

Monomorphisation

```
fn premier<T>(tab: [T; 10]) -> T {
    tab[0]
}
assert_eq!(premier([3u32, 4, 5]), 3u32);
assert_eq!(premier([3.5f64, 4., 5.2]), 3.5f64);
```

- chaque instantiation d'une fonction/méthode générique duplique le code
- des avantages
  - le code final est rapide
  - pas d'indirections
- des inconvénients
  - augmentation du temps de compilation
  - augmentation de la taille des programmes compilés
    - parfois au détriment des performances

Plan

Types algébriques

Types génériques

Structures génériques classiques

Les Tuples et Tableaux

```rust
fn premier_de_chaque_tableau<T, U>(
        t1: [T; 10],
        t2: [U; 10]
) -> (T, U) {
    (t1[0], t2[0])
}
```

https://doc.rust-lang.org/stable/std/primitive.tuple.html

Type 'S' or '/' to search, '?' for more options...

Primitive Type tuple

Since 1.0.0

A finite heterogeneous sequence, `(T, U, ..)`.

La Range

```rust
use std::ops::Range;
let r1 = 1u8..10;
let r2 = 3u32..300;
fn dernier<T>(r: &Range<T>) -> T {
    r.end
}
assert_eq!(dernier(&r1), 10u8);
let r3:Range<char> = 'a'..'z';
assert_eq!(dernier(&r3), 'z');
```

std::ops

Struct Range

Since 1.0.0 · Source

```rust
pub struct Range<Idx> {
    pub start: Idx,
    pub end: Idx,
}
```

L'Option

```rust
let o: Option<u32> = Some(3);
let o2 = Some(o);
fn unwrap<T>(o: Option<T>) -> T {
    match o {
        Some(t) => t,
        None => panic!("boom")
    }
}
assert_eq!(unwrap(unwrap(o2)), 3);
```

std::option

Enum Option

Since 1.0.0 · Source

```rust
pub enum Option<T> {
    None,
    Some(T),
}
```

L'Option

```
let o: Option<u32> = Some(3);
let o2 = Some(o);
fn unwrap<T>(o: Option<T>) -> T {
    match o {
        Some(t) => t,
        None => panic!("boom")
    }
}
assert_eq!(unwrap(unwrap(o2)), 3);
```

- quel est le type de **o2** ?

std::option

Enum Option

Since 1.0.0 · Source

```
pub enum Option<T> {
    None,
    Some(T),
}
```

L'Option

```
let o: Option<u32> = Some(3);
let o2 = Some(o);
fn unwrap<T>(o: Option<T>) -> T {
    match o {
        Some(t) => t,
        None => panic!("boom")
    }
}
assert_eq!(unwrap(unwrap(o2)), 3);
```

- quel est le type de **o2** ?
  - `Option<Option<u32>>`

std::option

Enum Option

Since 1.0.0 · Source

```
pub enum Option<T> {
    None,
    Some(T),
}
```

La Boite

- `Box<T>`
- objet alloué dans le tas
  - malloc à la création
  - free à la destruction
  - jamais de pointeur null
- `Option<Box<T>>`
  - optimisé à la compilation
    - même taille qu'un pointeur C

std::boxed

Struct Box

Since 1.0.0 · Source

```
pub struct Box<T, A = Global>(/* private fields */)
```

```
let e:Box<u32> = Box::new(3); // entier dans le tas
```

Le Vecteur

- `Vec<T>`
- tableau re-dimensionnable

```
impl<T> Vec<T> {
  fn push(&mut self, value: T) {
      ...
  }
  fn pop(&mut self) -> Option<T> {
      ...
  }
  ...
}
```

- macro `vec!`
  - `let v:Vec<_> = vec![1,2,3];`

std::vec

Struct Vec

Since 1.0.0 · Source

```
pub struct Vec<T, A = Global>
where
    A: Allocator,
{ /* private fields */ }
```