Const Generics

Frédéric Wagner

2025

Ensimag

Tableaux

- rust du passé

PartialEq<&'b [B]>
PartialEq<&'b mut [B]>
PartialEq<[B; 0]>
PartialEq<[B; 1]>
PartialEq<[B; 10]>
PartialEq<[B; 11]>
PartialEq<[B; 12]>
PartialEq<[B; 13]>
PartialEq<[B; 14]>
PartialEq<[B; 15]>
PartialEq<[B; 16]>
PartialEq<[B; 17]>
PartialEq<[B; 18]>
PartialEq<[B; 19]>
PartialEq<[B; 2]>

Primitive Type array

[−] A fixed-size array, denoted `[T; N]`, for the element type, `T`, and the non-negative compile-time constant size, `N`.

There are two syntactic forms for creating an array:

- A list with each element, i.e., `[x, y, z]`.
- A repeat expression `[x; N]`, which produces an array with `N` copies of `x`. The type of `x` must be `Copy`.

Arrays of sizes from 0 to 32 (inclusive) implement the following traits if the element type allows it:

- `Debug`
- `IntoIterator` (implemented for `&[T; N]` and `&mut [T; N]`)
- `PartialEq`, `PartialOrd`, `Eq`, `Ord`
- `Hash`
- `AsRef`, `AsMut`
- `Borrow`, `BorrowMut`
- `Default`

Tableaux

- rust du passé

PartialEq<&'b [B]>
PartialEq<&'b mut [B]>
PartialEq<[B; 0]>
PartialEq<[B; 1]>
PartialEq<[B; 10]>
PartialEq<[B; 11]>
PartialEq<[B; 12]>
PartialEq<[B; 13]>
PartialEq<[B; 14]>
PartialEq<[B; 15]>
PartialEq<[B; 16]>
PartialEq<[B; 17]>
PartialEq<[B; 18]>
PartialEq<[B; 19]>
PartialEq<[B; 2]>

Primitive Type array

[−] A fixed-size array, denoted `[T; N]`, for the element type, `T`, and the non-negative compile-time constant size, `N`.

There are two syntactic forms for creating an array:

- A list with each element, i.e., `[x, y, z]`.
- A repeat expression `[x; N]`, which produces an array with `N` copies of `x`. The type of `x` must be `Copy`.

Arrays of sizes from 0 to 32 (inclusive) implement the following traits if the element type allows it:

- `Debug`
- `IntoIterator` (implemented for `&[T; N]` and `&mut [T; N]`)
- `PartialEq`, `PartialOrd`, `Eq`, `Ord`
- `Hash`
- `AsRef`, `AsMut`
- `Borrow`, `BorrowMut`
- `Default`

- implémentations à coup de macros

Const generics

- paramètre générique qui est une valeur et non un type
  - souvent une taille
- sur les structures

```rust
struct Mouvement<const S: usize> {
    position: [u32; S],
}
```

- sur les fonctions

```rust
fn nouveau<const S: usize>() -> Mouvement<S> {
    Mouvement { position: [0; S] }
}
```

Limitations

```rust
struct Mouvement<const S: usize> {
    position: [u32; S],
    vitesses: [u32; S-1],
}
```

```
error: generic parameters may not be used in const operations
 --> src/main.rs:3:21
  |
3 |     vitesses: [u32; S - 1],
  |                     ^ cannot perform const operation using `S`
  |
  = help: const parameters may only be used as standalone arguments, i.e. `S`
  = help: add `#![feature(generic_const_exprs)]` to allow generic const expressions
```

Diagnostic

- quid de $S - 1$ si $S = 0$?
- ou $S + 1$ si `S=usize::MAX` ?
- quid de Mouvement$<\{\text{loop}\{\}\}>$?

Diagnostic

- quid de $S - 1$ si $S = 0$?
- ou $S + 1$ si `S=usize::MAX` ?
- quid de Mouvement$<\{$loop$\{\}\}>$?
- `https://hackmd.io/OsVAAEgOS2ajDwxg44kN-A?view`
  - pas près d'être résolu

Diagnostic

- quid de $S - 1$ si $S = 0$?
- ou $S + 1$ si `S=usize::MAX` ?
- quid de $\mathrm{Mouvement}<\{\mathrm{loop}\{\}\}>$?
- `https://hackmd.io/OsVAAEgOS2ajDwxg44kN-A?view`
  - pas près d'être résolu
- revient à avoir un moteur de preuve
  - `https://www.youtube.com/watch?v=BdXWlQsd7RI`
- 'solution' temporaire avec typenum et generic array

n reines

```
fn n_reines<const N: usize>() -> usize {
let mut echiquier = [[false; N]; N];
  fn n_reines_rec<const N: usize>(e: &mut [[bool; N]; N],
    l: usize) -> usize {
    if l == 0 { 1 } else {
      (0..N).filter_map(|c| {
          if !conflit(e, l - 1, c) {
            e[c][l - 1] = true;
            let compte = n_reines_rec(e, l - 1);
            e[c][l - 1] = false;
            Some(compte)
          } else { None }
        }).sum()
    }
  }
  n_reines_rec(&mut echiquier, N)
}
```