Monomorphisation

Frédéric Wagner

2025

Ensimag

Plan

Encore des itérateurs

Monomorphisation

Mesures de performances

successors

- calcule le suivant à partir du courant

```
pub fn successors<T, F>(first: Option<T>, succ: F)
        -> Successors<T, F>
where
    F: FnMut(&T) -> Option<T>,
```

- exemple : fibo

```
std::iter::successors(Some((0, 1)),
  |(a, b)| Some((*b, a+b)))
.map(|(a, _)| a)
```

from_fn

- écriture d'itérateurs complexes

```
    pub fn from_fn<T, F>(f: F)
        -> FromFn<F>
where
    F: FnMut() -> Option<T>,
```

from_fn

- écriture d'itérateurs complexes

```
pub fn from_fn<T, F>(f: F)
    -> FromFn<F>
where
    F: FnMut() -> Option<T>,
```

- chaque appel à la fonction renvoie l'élément suivant

from_fn

- écriture d'itérateurs complexes

```
    pub fn from_fn<T, F>(f: F)
        -> FromFn<F>
where
    F: FnMut() -> Option<T>,
```

- chaque appel à la fonction renvoie l'élément suivant
- exemple : fibo

```
let mut a = 0;
let mut b = 1;
std::iter::from_fn(move || {
  let r = a;
  a = b;
  b += r;
  Some(r)
})
```

Plan

Encore des itérateurs

Monomorphisation

Mesures de performances

Monomorphisation

- passage du générique au concret
  - en fixant les types génériques
  - re-génération du code
    - meilleures performances
    - augmentation des temps de compilation
    - duplication du code assembleur

Duplication

```
fn somme<N: AsRef<Path>>(
  nom_fichier: N,
) -> std::io::Result<u32> {
  let r = BufReader::new(File::open(nom_fichier)?);
  Ok(
    r.lines()
      .map_while(|l| l.ok())
      .filter_map(|l| l.parse::<u32>().ok())
      .sum(),
  )
}
```

Duplication

```
fn somme<N: AsRef<Path>>(
  nom_fichier: N,
) -> std::io::Result<u32> {
  let r = BufReader::new(File::open(nom_fichier)?);
  Ok(
    r.lines()
      .map_while(|l| l.ok())
      .filter_map(|l| l.parse::<u32>().ok())
      .sum(),
  )
}
fn somme2<N: AsRef<Path>>(
  nom_fichier: N,
) -> std::io::Result<u32> {
  let f = File::open(nom_fichier)?;
  Ok(somme_fichier(f))
}
```

Itérateurs

- la monomorphisation est critique pour les performances
- exemple : produit scalaire

```
a.iter().zip(&b).map(|(ea, eb)| *ea * *eb).sum()
```

Itérateurs

- la monomorphisation est critique pour les performances
- exemple : produit scalaire

```
a.iter().zip(&b).map(|(ea, eb)| *ea * *eb).sum()
```

- type de l'itérateur

```
Map<Zip<Iter, Iter>, impl FnMut((&i32, &i32)) -> i32>
```

Itérateurs

- la monomorphisation est critique pour les performances
- exemple : produit scalaire

```
a.iter().zip(&b).map(|(ea, eb)| *ea * *eb).sum()
```

- type de l'itérateur

```
Map<Zip<Iter, Iter>, impl FnMut((&i32, &i32)) -> i32>
```

- pourtant les performances sont super

Itérateurs

- la monomorphisation est critique pour les performances
- exemple : produit scalaire

```
a.iter().zip(&b).map(|(ea, eb)| *ea * *eb).sum()
```

- type de l'itérateur

```
Map<Zip<Iter, Iter>, impl FnMut((&i32, &i32)) -> i32>
```

- pourtant les performances sont super
- on peut aider le compilateur avec quelques astuces

fold du Map

```
  fn fold<Acc, G>(self, init: Acc, g: G) -> Acc
    where G: FnMut(Acc, Self::Item) -> Acc,
  {
      self.iter.fold(init, map_fold(self.f, g))
  }
  fn map_fold<T, B, Acc>(
    mut f: impl FnMut(T) -> B,
    mut g: impl FnMut(Acc, B) -> Acc,
) -> impl FnMut(Acc, T) -> Acc {
    move |acc, elt| g(acc, f(elt))
  }
```

- la composition améliore l'inlining
- appels des **fold** sous-jacents optimisés

fold de Iter (1/2)

```
fn fold<B, F>(self, init: B, mut f: F) -> B
    where
        F: FnMut(B, Self::Item) -> B,
{
    // this implementation consists of the following
    // optimizations compared to the default implementation:
    // - do-while loop, as is llvm's preferred loop shape,
    //   see https://releases.llvm.org/16.0.0/docs/LoopTermino
    // - bumps an index instead of a pointer since the latter
    // case inhibits some optimizations, see #111603
    // - avoids Option wrapping/matching
```

fold de Iter (2/2)

```
    if is_empty!(self) {
        return init;
    }
    let mut acc = init;
    let mut i = 0;
    let len = len!(self);
    loop {
        // SAFETY: the loop iterates `i in 0..len`, which
        // always is in bounds of the slice allocation
        acc = f(acc, unsafe { & $( $mut_ )? *self.ptr.add(i).a
        i = unsafe { i.unchecked_add(1) };
        if i == len {
            break;
        }
    }
    acc
}
```

size_hint

- size_hint
  - `fn size_hint(&self) -> (usize, Option<usize>)`
  - renvoie une borne inférieure et une borne supérieure sur le nombre d'itérations restantes

size_hint

- size_hint
  - `fn size_hint(&self) -> (usize, Option<usize>)`
  - renvoie une borne inférieure et une borne supérieure sur le nombre d'itérations restantes
  - permet différentes optimisations

size_hint

- size_hint
  - `fn size_hint(&self) -> (usize, Option<usize>)`
  - renvoie une borne inférieure et une borne supérieure sur le nombre d'itérations restantes
  - permet différentes optimisations

```
fn collect_to_vec<I:Iterator>(i: I) -> Vec<I::Item> {
  let n = i.size_hint().1.unwrap_or_default();
  i.fold(Vec::with_capacity(n), |mut v, e| {
    v.push(e);
    v
  })
}
```

Plan

Encore des itérateurs

Monomorphisation

Mesures de performances

Mesurer les performances

- un art délicat
- surtout pour des micro-benchmarks : de nombreux pièges
  - défauts de cache
  - défauts de page
  - alignement du code
  - perturbations extérieures
  - optimisations du compilateur
  - ...
- à compléter avec un regard sur le code assembleur

Mesures manuelles

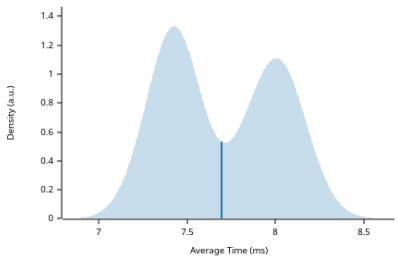
- `std::time::Instant::now`
  - pas cher (quelques nano secondes)
- `std::hint::black_box`
  - évite certaines optimisations

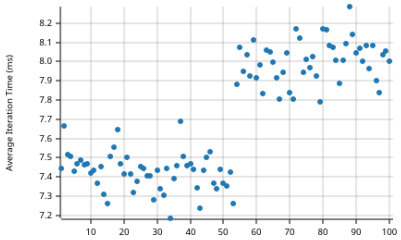
```
const N: u64 = 100_000_000;

fn recherche<T: Eq, I: Iterator<Item = T>>(i: I, x: T) -> bool
    std::hint::black_box(i).any(|e| e == x)
}

fn main() {
    let debut = std::time::Instant::now();
    assert!(recherche(0..N, N / 2));
    println!("{:?}", debut.elapsed());
}
```

Benchmarks

- **cargo bench**
- criterion
  - un peu de stats
  - guide en ligne
  - rapport complet dans target/criterion

un-even input/generique-opt





Additional Statistics:

	Lower bound	Estimate	Upper bound
R^2	0.0094790	0.0098356	0.0094665
Mean	7.6376 ms	7.6971 ms	7.7577 ms
Std. Dev.	284.72 µs	306.91 µs	325.39 µs
Median	7.4694 ms	7.6013 ms	7.8799 ms
MAD	217.25 µs	391.14 µs	463.98 µs

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

Understanding this report:

The plot on the left displays the average time per iteration for this benchmark. The shaded region shows the estimated probability of an iteration taking a certain amount of time, while the line shows the mean. Click on the plot for a larger view showing the outliers.

The plot on the right shows the average time per iteration for the samples. Each point represents one sample.

See the documentation for more details on the additional statistics.