Surcharge, traits et type classes

Programmation avancée

Jacques-Henri Jourdan

14 mars 2025

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

1 Introduction

2 Surcharge en Java

3 Type traits
- Intro : généricité et surcharge
- Premiers exemples
- Surcharge d'opérateurs
- Itérateurs
- Contrats de traits
- Héritage
- Cohérence

4 Typeclasses de Haskell

La surcharge

Qu'est-ce que c'est ?

Introduction

Surcharge en Java

Type traits

Intro : généricité et surcharge

Premiers exemples

Surcharge d'opérateurs

Itérateurs

Contrats de traits

Héritage

Cohérence

Typeclasses de Haskell

La surcharge

Surcharge (*overloading*) : lorsque un nom fait référence à plusieurs définitions simultanément.

Par exemple, dans beaucoup de langages, les opérateurs arithmétiques sont surchargés : Ils peuvent être utilisés avec plusieurs types numériques.

Exemple, en C :

```c
int main() {
    int a = 12, b = 13;
    printf("%d\n", a + b); // Addition sur les entiers

    double c = 12.3, d = 13.5;
    printf("%f\n", c + d); // Addition sur les nombres à virgule flottante

    return 0;
}
```

Ici, c'est le type des opérandes qui permet la désambiguïsation.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

La surcharge

Surcharge (*overloading*) : lorsque un nom fait référence à plusieurs définitions simultanément.

Parfois, c'est le nombre de paramètres qui permet la désambiguïsation.

Exemple, en C++ :

```
int max(int a, int b) {
    return a < b ? b : a;
}

int max(int a, int b, int c) {
    return max(a, max(b, c));
}

int max(double a, double b) {
    return a < b ? b : a;
}

int main() {
    cout << max(1, 2, 3) << endl:
    cout << max(1., 2.) << endl:
}
```


Introduction

Surcharge en Java

Type traits

Intro : généricité et surcharge

Premiers exemples

Surcharge d'opérateurs

Itérateurs

Contrats de traits

Héritage

Cohérence

Typeclasses de Haskell

La surcharge

Surcharge (*overloading*) : lorsque un nom fait référence à plusieurs définitions simultanément.

Enfin, ce peut être le type de retour qui permet la désambiguïsation.

Exemple, en Rust :

```rust
fn main() {
    let x: i32 = Default::default();          // == 0
    let y: Option<i32> = Default::default();  // == None

    println!("{:?} {:?}", x, y);
}
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

La surcharge, pourquoi ?

Permet d'alléger les notations en utilisant le même symbole ou nom pour des concepts proches.

Introduction

Surcharge en Java

Type traits

Intro : généricité et surcharge

Premiers exemples

Surcharge d'opérateurs

Itérateurs

Contrats de traits

Héritage

Cohérence

Typeclasses de Haskell

La surcharge, pourquoi ?

Permet d'alléger les notations en utilisant le même symbole ou nom pour des concepts proches.

Dans certains langages, la surcharge permet une forme de programmation générique. I.e., écrire le même code pour opérer sur des valeurs de types différents.

Exemple, en C++ :

```
int max(int a, int b) { ... }
int max(double a, double b) { ... }

template <typename T>
T max(T a, T b, T c) {
  return max(a, max(b, c));
}

int main() {
  cout << max(1, 2, 3) << endl;    // max est utilisé pour des entiers
  cout << max(1., 2., 3.) << endl; // max est utilisé pour des doubles
}
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Ne pas confondre

Shadowing

La surcharge est à distinguer du *shadowing*, où une variable est liée à une variable différente selon le contexte.

Exemple en Rust :

```
let x = 12;
{ let x = 13;
  println!("{}", x);  // Affiche 13
}
println!("{}", x);    // Affiche 12
```

Le nom x est lié à 12 ou à 13, selon le contexte.
Il n'y a pas d'ambiguïté : pour un point de programme, la variable x n'est liée qu'à une seule valeur.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Y a-t-il de la surcharge en OCaml ?

Introduction

Surcharge en Java

Type traits

Intro : généricité et surcharge

Premiers exemples

Surcharge d'opérateurs

Itérateurs

Contrats de traits

Héritage

Cohérence

Typeclasses de Haskell

Y a-t-il de la surcharge en OCaml ?

```
# type r1 = { f1: int; f: int };;
# type r2 = { f2: int; f: int };;
# let f x = x.f;;
val f : r2 -> int = <fun>
# let g x = (x.f1, x.f);;
val g : r1 -> int * int = <fun>
```

Le nom de champ `f` est surchargé !
(Un comportement similaire existe pour les noms de constructeurs d'ADTs.)

La résolution de cette surcharge se fait en fonction des informations connues par l'inférence de type au point d'utilisation.
$\Longrightarrow$ dépend des mécanismes internes de l'inférence.

De manière générale, l'interaction entre inférence et surcharge est compliquée.
Voilà pourquoi + et +. sont deux notations différentes en OCaml.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Y a-t-il de la surcharge en OCaml ?

```
# type r1 = { f1: int; f: int };;
# type r2 = { f2: int; f: int };;
# let f x = x.f;;
val f : r2 -> int = <fun>
# let g x = (x.f1, x.f);;
val g : r1 -> int * int = <fun>
```

Le nom de champ `f` est surchargé !
(Un comportement similaire existe pour les noms de constructeurs d'ADTs.)

La résolution de cette surcharge se fait en fonction des informations connues par l'inférence de type au point d'utilisation.
$\Longrightarrow$ dépend des mécanismes internes de l'inférence.

De manière générale, l'interaction entre inférence et surcharge est compliquée.
Voilà pourquoi + et +. sont deux notations différentes en OCaml.

Note : une extension d'OCaml, les modular implicits, propose une notion de surcharge plus puissante, proche des type traits de Rust.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Résolution de la surcharge

Il est nécessaire d'avoir des règles qui précisent comment la surcharge est résolue.

Deux questions à répondre pour une référence à un nom surchargé :

1. Quelles sont les définitions candidates ?
   - Ex : bonne visibilité, bon nombre d'argument, types d'arguments compatibles, ...
2. Parmi celles-ci, quelles est la meilleure ?
   - Ex : dernière définition (champ de records en OCaml), types d'arguments « les plus proches »...

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

1 Introduction

2 Surcharge en Java

3 Type traits
- Intro : généricité et surcharge
- Premiers exemples
- Surcharge d'opérateurs
- Itérateurs
- Contrats de traits
- Héritage
- Cohérence

4 Typeclasses de Haskell

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

La surcharge en Java

Java propose un mécanisme de surcharge statique
(i.e., résolue uniquement avec des information connues à la compilation).

Exemple :

```
class Main {
      static void print(int a) { ... }
      static void print(String s) { ... }

      public static void main() {
           print(12);
      }
}
```


Introduction

Surcharge en Java

Type traits

Intro : généricité et surcharge

Premiers exemples

Surcharge d'opérateurs

Itérateurs

Contrats de traits

Héritage

Cohérence

Typeclasses de Haskell

La surcharge en Java

Java propose un mécanisme de surcharge statique
(i.e., résolue uniquement avec des information connues à la compilation).

Parfois, à cause de l'héritage, plusieurs méthodes peuvent être utilisées :

```
class A { ... }
class B extends A { ... }  // La classe ‘B‘ hérite de la classe ‘A‘.
                           // Tout objet de type ‘B‘ est aussi de type ‘A‘.

class Main {
      static void f(A x, B y) { ... }
      static void f(B x, A y) { ... }


      static void g(A a, B b) {
          f(a, b);  // Un seul candidat possible.
          f(b, a);  // idem.
          f(b, b);  // Deux candidats, aucun n'est plus spécifique.
                    // ==> ERREUR
      }
}
```

Surcharge, traits et type classes
Programmation avancée
Jacques-Henri Jourdan
Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

La surcharge en Java

Java propose un mécanisme de surcharge statique
(i.e., résolue uniquement avec des information connues à la compilation).

Parfois, à cause de l'héritage, plusieurs méthodes peuvent être utilisées :

```
class A { ... }
class B extends A { ... }  // La classe ‘B‘ hérite de la classe ‘A‘.
                           // Tout objet de type ‘B‘ est aussi de type ‘A‘.

class Main {
      static void f(A x, B y) { ... }
      static void f(B x, A y) { ... }
      static void f(B x, B y) { ... }

      static void g(A a, B b) {
          f(a, b);  // Un seul candidat possible.
          f(b, a);  // idem.
          f(b, b);  // Trois candidats, l'un d'eux est plus spécifique que tous les autres.
                    // ==> OK, on prend ce meilleur candidat.
      }
}
```

Surcharge, traits et type classes
Programmation avancée
Jacques-Henri Jourdan
Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

La surcharge en Java

Java propose un mécanisme de surcharge statique
(i.e., résolue uniquement avec des information connues à la compilation).

On utilise uniquement le type statique (pour une résolution à la compilation) :

```
class A { ... }
class B extends A { ... }  // La classe ‘B‘ hérite de la classe ‘A‘.
                           // Tout objet de type ‘B‘ est aussi de type ‘A‘.

class Main {
      static void h(A x) { ... }
      static void h(B x) { ... }

      public static void main() {
          A x = new B(); // ‘x‘ contient un objet de type ‘B‘, mais son *type statique* est ‘A‘.
          h(x)  // La surcharge est résolue en fonction du type statique.
                // ==> C’est ‘static void h(A a) { ... }‘ qui est appelée,
                //     même si le type dynamique (à l’exécution) de ‘x‘ est ‘B‘
      }
}
```

Ne pas confondre

Redéfinition (*overriding*)

En java (comme dans tout langage orienté objet), il est possible de redéfinir (*override*) une méthode lorsque l'on hérite d'une classe.
Ces méthodes sont dites virtuelles.

La redéfinition peut être vue comme de la surcharge au sens de la définition donnée au début du cours. (Plusieurs définitions visibles pour un même nom.)

Mais quand on travaille dans un langage orienté objet, le terme "surcharge" ne réfère jamais à la redéfinition des méthodes virtuelles (overriding).

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Ne pas confondre

Redéfinition (*overriding*)

Dans les langages orientés objets :

- La surcharge est résolue statiquement,
  - avec le type statique (i.e., à la compilation) des paramètres, et le nombre d'arguments.
- Un appel à une méthode virtuelle est résolu dynamiquement.
  - En utilisant le type dynamique (i.e., à l'exécution) de l'objet.

Cette subtilité est une source fréquente de confusion !

Note : nous ferons un cours spécifique sur la programmation orientée objet pour bien comprendre les mécanismes d'héritage et de redéfinition.

Introduction

Surcharge en Java

Type traits

Intro : généricité et surcharge

Premiers exemples

Surcharge d'opérateurs

Itérateurs

Contrats de traits

Héritage

Cohérence

Typeclasses de Haskell

1 Introduction

2 Surcharge en Java

3 Type traits
- Intro : généricité et surcharge
- Premiers exemples
- Surcharge d'opérateurs
- Itérateurs
- Contrats de traits
- Héritage
- Cohérence

4 Typeclasses de Haskell

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

La surcharge : un outil pour la généricité

La surcharge en Java n'est qu'un mécanisme de simplification des notations.

Mais en C++, on peut écrire, par exemple :

```
template <typename T> T sum_seq(const vector<T> &v) {
  T res = v[0];
  for(size_t i = 1; i < v.size(); i++)
    res = res + v[i];    // Utilisation de + sur le type T
  return res;
}
```

```
int main() {
    cout << sum_seq(vector<int>({1, 2, 3, 4, 5})) << endl;

}
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

La surcharge : un outil pour la généricité

La surcharge en Java n'est qu'un mécanisme de simplification des notations.

Mais en C++, on peut écrire, par exemple :

```
template <typename T> T sum_seq(const vector<T> &v) {
  T res = v[0];
  for(size_t i = 1; i < v.size(); i++)
    res = res + v[i];    // Utilisation de + sur le type T
  return res;
}

struct vec5 { double t[5]; }; // Type de vecteur de longueur 5
vec5 operator+(const vec5 &x, const vec5 &y) {  // Surcharge de l'opérateur '+' pour vec5
  vec5 res;
  for(int i = 0; i < 5; i++)
      res.t[i] = x.t[i] + y.t[i];
  return res;
}

int main() {
    cout << sum_seq(vector<int>({1, 2, 3, 4, 5})) << endl;
    vector<vec5> vecs({ {{1.,2.,0.,3.1,12.}},   {{1.1,2.8,1.2,-2.4,0.}} });
    vec5 vecsum = sum_seq(vecs);
}
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

La surcharge : un outil pour la généricité

La surcharge en Java n'est qu'un mécanisme de simplification des notations.

Mais en C++, on peut écrire, par exemple :

```
template <typename T> T sum_seq(const vector<T> &v) {
  T res = v[0];
  for(size_t i = 1; i < v.size(); i++)
    res = res + v[i];    // Utilisation de + sur le type T
  return res;
}

struct vec5 { do
vec5 operator+(c
  vec5 res;
  for(int i = 0; i < 5; i++)
      res.t[i] = x.t[i] + y.t[i];
  return res;
}

int main() {
    cout << sum_seq(vector<int>({1, 2, 3, 4, 5})) << endl;
    vector<vec5> vecs({ {{1.,2.,0.,3.1,12.}},   {{1.1,2.8,1.2,-2.4,0.}} });
    vec5 vecsum = sum_seq(vecs);
}
```

La fonction `sum_seq` est générique : elle fonctionne pour tout type `T` pour lequel l'opérateur + est implémenté.

Abstraction cassée

```
template <typename T> T sum_seq(const vector<T> &v) {
  T res = v[0];
  for(size_t i = 1; i < v.size(); i++)
    res = res + v[i];
  return res;
}
```

Le prototype (i.e., le type) de `sum_seq` ne renseigne pas sur la nécessité que l'opérateur + soit surchargé pour `T`.
Seul le corps de `sum_seq` le dit. Cest une rupture d'abstraction.

Un premier problème est que les messages d'erreurs sont souvent très mauvais.
(La vérification des types d'une fonction générique en C++ n'est effectuée qu'à l'instantiation.)

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Abstraction coûtée

```
template <typenar
  T res = v[0];
  for(size_t i =
    res = res +
  return res;
}
```

Le prototype (i.
soit surchargé p
Seul le corps de

Un premier prob
(La vérification
l'instantiation.)

Exemple : si j'utilise une table de hachage en C++ en oubliant de fournir une fonction de hachage :

```
test.cpp: Dans la fonction « int main() »:
test.cpp:31:25: erreur: utilisation de la fonction supprimée « std::unordered_set<_Value, _Hash, _Pred, _Alloc>::unordered_set() [avec _Va
   31 |     unordered_set<vec5> h;
      |                         ^
Dans le fichier inclus depuis /usr/include/c++/11/unordered_set:47,
                 depuis test.cpp:3:
/usr/include/c++/11/bits/unordered_set.h:135:7: note: « std::unordered_set<_Value, _Hash, _Pred, _Alloc>::unordered_set() [avec _Value = v
  135 |       unordered_set() = default;
      |       ^~~~~~~~~~~~~
/usr/include/c++/11/bits/unordered_set.h:135:7: erreur: utilisation de la fonction supprimée « std::_Hashtable<_Key, _Value, _Alloc, _Extra
Dans le fichier inclus depuis /usr/include/c++/11/unordered_set:46,
                 depuis test.cpp:3:
/usr/include/c++/11/bits/hashtable.h:528:7: note: « std::_Hashtable<_Key, _Value, _Alloc, _ExtractKey, _Equal, _Hash, _RangeHash, _Unused,
  528 |       _Hashtable() = default;
      |       ^~~~~~~~~
/usr/include/c++/11/bits/hashtable.h:528:7: erreur: utilisation de la fonction supprimée « std::__detail::_Hashtable_base<_Key, _Value, _E
Dans le fichier inclus depuis /usr/include/c++/11/bits/hashtable.h:35,
                 depuis /usr/include/c++/11/unordered_set:46,
                 depuis test.cpp:3:
/usr/include/c++/11/bits/hashtable_policy.h:1604:7: note: « std::__detail::_Hashtable_base<_Key, _Value, _ExtractKey, _Equal, _Hash, _Range
 1604 |       _Hashtable_base() = default;
      |       ^~~~~~~~~~~~~~
/usr/include/c++/11/bits/hashtable_policy.h:1604:7: erreur: utilisation de la fonction supprimée « std::__detail::_Hash_code_base<_Key, _V
/usr/include/c++/11/bits/hashtable_policy.h: Dans l'instanciation de « std::__detail::_Hashtable_ebo_helper<_Nm, _Tp, true>::_Hashtable_eb
/usr/include/c++/11/bits/hashtable_policy.h:1210:7:   requis depuis ici
/usr/include/c++/11/bits/hashtable_policy.h:1127:49: erreur: utilisation de la fonction supprimée « std::hash<vec5>::hash() »
 1127 |       _Hashtable_ebo_helper() noexcept(noexcept(_Tp())) : _Tp() { }
      |                                                 ^~~~~
[Nombreuses lignes coupées]
```

Introduction

Surcharge en Java

Intro : généricité et surcharge

Surcharge d'opérateurs

Contrats de traits

Héritage

Typeclasses de Haskell

Abstraction cassée

```
template <typename T> T sum_seq(const vector<T> &v) {
  T res = v[0];
  for(size_t i = 1; i < v.size(); i++)
    res = res + v[i];
  return res;
}
```

Le prototype (i.e., le type) de `sum_seq` ne renseigne pas sur la nécessité que l'opérateur + soit surchargé pour `T`.
Seul le corps de `sum_seq` le dit. Cest une rupture d'abstraction.

Un premier problème est que les messages d'erreurs sont souvent très mauvais.
(La vérification des types d'une fonction générique en C++ n'est effectuée qu'à l'instantiation.)

Pire, si la fonction surchargée existe mais n'a pas le bon type, la compilation peut réussir, mais avec un comportement inattendu.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Les type traits en Rust

En Rust, la surcharge est contrôlée par des type traits, qui décrivent ce qu'on peut attendre d'une fonction surchargée.

Exemple : le trait `Clone` déclare une méthode « surchargeable » `clone` pour copier en dupliquant l'ownership.

```
// Dans std::clone
trait Clone {
    fn clone(&self) -> Self;
}
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Les type traits en Rust

En Rust, la surcharge est contrôlée par des type traits, qui décrivent ce qu'on peut attendre d'une fonction surchargée.

Exemple : le trait `Clone` déclare une méthode « surchargeable » `clone` pour copier en dupliquant l'ownership.

```
// Dans std::clone
trait Clone {
    fn clone(&self) -> Self;
}
```

Un type d'arbre :

```
enum Tree { Leaf, Node(Box<Tree>, i64, Box<Tree>) }
use Tree::*;
```

On "instancie" `Clone` pour `Tree` :

```
impl Clone for Tree {
    fn clone(&self) -> Self {
        match self {
            Leaf => Leaf,
            Node(box b1, x, box b2) =>
                Node(Box::new(b1.clone()),
                     *x,
                     Box::new(b2.clone()))
        }
    }
}
```


Surcharge, traits et type classes

Programmation avancée

Jacques-Henri Jourdan

Introduction

Surcharge en Java

Type traits

Intro : généricité et surcharge

Premiers exemples

Surcharge d'opérateurs

Itérateurs

Contrats de traits

Héritage

Cohérence

Typeclasses de Haskell

Les type traits en Rust

En Rust, la surcharge est contrôlée par des type traits, qui décrivent ce qu'on peut attendre d'une fonction surchargée.

Exemple : le trait `Clone` déclare une méthode « surchargeable » `clone` pour copier en dupliquant l'ownership.

```
// Dans std::clone
trait Clone {
    fn clone(&self) -> Self;
}
```

On peut ensuite utiliser `clone` pour `Tree`

```
fn cloned_pair(t: Tree) -> (Tree, Tree) {
    let t2 = t.clone();
    (t, t2)
}
```


Surcharge, traits et type classes

Programmation avancée

Jacques-Henri Jourdan

Introduction

Surcharge en Java

Type traits

Intro : généricité et surcharge

Premiers exemples

Surcharge d'opérateurs

Itérateurs

Contrats de traits

Héritage

Cohérence

Typeclasses de Haskell

Type traits et programmation générique

Mais on peut aussi faire une version générique de la fonction `cloned_pair` :

```
fn cloned_pair<T>(t: T) -> (T, T)
   where T: Clone
{
    // L'appel à clone() est autorisé parce
    // qu'on a déclaré T: Clone
    let t2 = t.clone();
    (t, t2)
}
```

On rajoute une contrainte « `T: Clone` » au polymorphisme :

- peut être utilisée dans le corps de la fonction,
- devra être satisfaite au point d'appel.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Type traits et programmation générique

Mais on peut aussi faire une version générique de la fonction `cloned_pair` :

```
// Syntaxe simplifiée pour les contraintes simples.
fn cloned_pair<T: Clone>(t: T) -> (T, T)

{
    // L'appel à clone() est autorisé parce
    // qu'on a déclaré T: Clone
    let t2 = t.clone();
    (t, t2)
}
```

On rajoute une contrainte « `T: Clone` » au polymorphisme :

- peut être utilisée dans le corps de la fonction,
- devra être satisfaite au point d'appel.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Instances génériques

D'ailleurs, on peut aussi créer des instances génériques, potentiellement contraintes :

```
enum Tree<T> { Leaf, Node(Box<Tree>, T, Box<Tree>) }
use Tree::*;

impl<T: Clone> Clone for Tree<T> {
     fn clone(&self) -> Self {
          match self {
              Leaf => Leaf,
              Node(box b1, x, box b2) =>
                  Node(Box::new(b1.clone()),
                       x.clone(),
                       Box::new(b2.clone()))
          }
     }
}
```

Instances génériques

D'ailleurs, on peut aussi créer des instances génériques, potentiellement contraintes :

```
enum Tree<T> { Leaf, Node(Box<Tree>, T, Box<Tree>) }
use Tree::*;

impl<T: Clone> Clone for Tree<T> {
     fn clone(&self) -> Self {
          match self {
              Leaf => Leaf,
              Node(box b1, x, box b2) =>
                  Node(Box::new(b1.clone()),
                       x.clone(),
                       Box::new(b2.clone()))
          }
     }
}
```

Note : pour certains traits (dont `Clone`), les instances simples peuvent être générées automatiquement :

```
#[derive(Clone)]
enum Tree<T> { Leaf, Node(Box<Tree>, T, Box<Tree>) }
```

Implémentation par défaut

Exemple : le trait `Clone` a en fait une deuxième méthode :

```
// Dans std::clone
trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

La méthode `clone_from` fait comme `clone`, mais vers un objet déjà existant.

- La plupart du temps, on veut utiliser l'implémentation par défaut.
  - Faire le clone, désallouer l'ancien objet, puis y écrire le clone.
  - Rien à déclarer lors de l'instantiation.
- Certains types (comme `Vec`) ont une implémentation optimisée :
  - Implémentation spécifique lors de l'instanciation.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Deuxième exemple : trait Hash

Les bibliothèques de table de hachage de Rust nécessitent que le type des clefs implémente le trait `Hash` :

```
impl<K: Eq + Hash, V> HashMap<K, V> {
....
    fn insert(&mut self, k: K, v: V) -> Option<V> { ... }
....
}
```

La méthode `insert` est polymorphe par rapport aux types `K` (clefs) et `V` (valeurs).

Le polymorphisme est introduit au niveau du bloc `impl`
(pas forcément au niveau de la déclaration de la méthode).

On ajoute une contrainte `K: Eq + Hash` pour s'assurer que `K` est équipé de :

- une méthode `hash` pour hacher les clefs,
- un opérateur == pour les comparer.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Troisième exemple : trait `Default`

Pour certains types, on peut donner une « valeur par défaut », qui sert, par exemple, lorsqu'il faut initialiser une structure de donnée...

```
// Dans std::default
trait Default {
    fn default() -> Self;
}
```

```
fn main() {
    let x: i32 = Default::default();          // == 0
    let y: Option<i32> = Default::default();  // == None

    println!("{:?} {:?}", x, y);
}
```

De manière intéressante, c'est le type de retour qui permet de choisir la bonne instance.

Surcharge d'opérateurs

Exemple : opérateur « + »

```
// Dans std::ops :
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Le trait `Add` permet de surcharger « + ».

L'opérateur « + » n'est que du sucre syntaxique pour la méthode `add`.

Surcharge d'opérateurs

Exemple : opérateur « + »

```
// Dans std::ops :
trait Add<Rhs        > {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Le trait `Add` est paramétré :

- `Self` : type de l'opérande de gauche,
- `Rhs` : type de l'opérande de droite.

Les deux types sont utilisés lors de la recherche d'instance.

Surcharge d'opérateurs

Exemple : opérateur « + »

```
// Dans std::ops :
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

Le trait `Add` est paramétré :

- `Self` : type de l'opérande de gauche,
- `Rhs` : type de l'opérande de droite.

Les deux types sont utilisés lors de la recherche d'instance.

Note : par défaut, `Rhs = Self`.
Mais pas toujours :

```
impl<'a> Add<i32> for &'a i32 { ... }
impl<'a> Add<&'a i32> for i32 { ... }
...
```

Surcharge d'opérateurs

Exemple : opérateur « + »

```
// Dans std::ops :
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

`Add` contient un type associé, `Output`.

Permet de choisir un type de retour de « + » différent pour chaque paire (`Self`, `Rhs`).

Exemples :

```
impl<'a> Add<i32> for &'a i32 {
    type Output = i32; ...
}
impl Add<u64> for u64 {
    type Output = u64; ...
}
impl Add<Duration> for SystemTime {
    type Output = SystemTime; ...
}
```

Surcharge d'opérateurs

Exemple : opérateur « + »

```
// Dans std::ops :
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

`Add` contient un type associé, `Output`.

Permet de choisir un type de retour de « + » différent pour chaque paire (`Self`, `Rhs`).

Exemples :

```
impl<'a> Add<i32> for &'a i32 {
    type Output = i32; ...
}
impl Add<u64> for u64 {
    type Output = u64; ...
}
impl Add<Duration> for SystemTime {
    type Output = SystemTime; ...
}
```

C'est un composant d'un type trait (au même titre qu'une méthode).

Il ne sert pas à la désambiguïsation.
La recherche d'une instance appropriée ne prend en compte que `Self` et `Rhs`.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Autres opérateurs

En Rust, on peut surcharger tous les opérateurs :

- Arithmétiques : `Add`, `Sub`, `Mul`, `Neg`, ...
- Comparaison : `PartialEq`, `Eq`, `PartialOrd`, `Ord`
- Opération bit-à-bit : `BitAnd`, `BitOr`, `Not`, ...
- Opérations-mutations (`+=`, `*=`, ...) : `AddAssign`, `MulAssign`, ...
- Indexation (`x[y]`) : `Index`, `IndexMut`.
- Déréférentiation (`*x`) : `Deref`, `DerefMut`

Un autre exemple de trait : Iterator and IntoIterator

Permet d'énumérer les éléments d'une structure de donnée, d'un intervalle d'entiers...

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
  ...
}

trait IntoIterator {
  type Item;
  type IntoIter: Iterator<Item = Self::Item>; // Contrainte: le type associé Item de Self doit
                                              // être égal à celui de l'itérateur.
  fn into_iter(self) -> Self::IntoIter;
}
```

Sucre syntaxique :

```
for x in c {
  ...
}
```

$\Longrightarrow$

```
let mut it = c.into_iter();
loop {
  match it.next() {
    None => break,
    Some(x) => {
      ...
    }
  }
}
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Un autre exemple de trait : Iterator and IntoIterator

Permet d'énumérer les éléments d'une structure de donnée, d'un intervalle d'entiers...

```
trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
  ...
}

trait IntoI
  type Item
  type Into

  fn into_i
}
```

Important : `Iterator` n'est pas un type !

Il y a un type différent pour chaque implémentation de `Iterator`.
Ce type décrit l'état de l'itération, spécialisé pour le genre d'itérateur. Il diffère en fonction de la structure de donnée sur laquelle on itère.

Très différent du type `Seq.t` en OCaml (par exemple).

Sucre synta

```
for x in c {
  ...
}
```

$\Longrightarrow$

```
let mut it = c.into_iter();
loop {
  match it.next() {
    None => break,
    Some(x) => {
      ...
    }
  }
}
```

Intervalles

- Les intervalles tels que `0..10` implémentent `IntoIterator` et `Iterator`.
- `for i in 0..N {...}` est facilement optimisé.
  - C'est en fait le moyen idiomatique d'écrire des boucles `for`.

Intervalles

- Les intervalles tels que `0..10` implémentent `IntoIterator` et `Iterator`.
- `for i in 0..N {...}` est facilement optimisé.
  - C'est en fait le moyen idiomatique d'écrire des boucles `for`.

```
// Le type derrière la syntaxe 0..N (dans la bibliothèque standard de Rust)
// (Spécialisé à i32 pour la simplicité)
struct Range {
  start: i32,
  end: i32,
}

impl Iterator for Range {
  type Item = i32;

  fn next(&mut self) -> Option<i32> {
    if self.start < self.end {
      let r = self.start; self.start += 1;
      return Some(r)
    } else {
      return None
    }
  }
}
```

Intervalles

- Les intervalles tels que `0..10` implémentent `IntoIterator` et `Iterator`.
- `for i in 0..N {...}` est facilement optimisé.
  - C'est en fait le moyen idiomatique d'écrire des boucles `for`.

The code écrit par l'utilisateur :

```
for i in 0..N {
  ...
}
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Intervalles

- Les intervalles tels que `0..10` implémentent `IntoIterator` et `Iterator`.
- `for i in 0..N {...}` est facilement optimisé.
  - C'est en fait le moyen idiomatique d'écrire des boucles `for`.

Dépliage du sucre syntaxique :

```
let r = Range{ start: 0, end: N };
let mut it = r; // IntoIterator est trivial pour Range
loop {
  match it.next() {
    None => break,
    Some(i) => {
      ...
    }
  }
}
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Intervalles

- Les intervalles tels que `0..10` implémentent `IntoIterator` et `Iterator`.
- `for i in 0..N {...}` est facilement optimisé.
  - C'est en fait le moyen idiomatique d'écrire des boucles `for`.

On déplie la définition de `next` (*inlining*) :

```
let r = Range{ start: 0, end: N };
let mut it = r;
loop {
  let o =
    if it.start < end {
      let r = it.start; it.start += 1;
      Some(r)
    } else { None };
  match o {
    None => break,
    Some(i) => {
      ...
    }
  }
}
```

Intervalles

- Les intervalles tels que `0..10` implémentent `IntoIterator` et `Iterator`.
- `for i in 0..N {...}` est facilement optimisé.
  - C'est en fait le moyen idiomatique d'écrire des boucles `for`.

Après l'inversion de `if` et `match`, et simplifications :

```
let mut it = Range{ start: 0, end: N };
loop {
  if it.start < it.end {
    let i = it.start;
    it.start += 1;
    ...
  } else {
    break
  }
}
```

C'est très proche de ce qu'on peut attendre d'une boucle `for` en C/C++.
Il reste à :

- déplacer `it.start += 1` à la fin de la boucle ;
- placer `it.start` et `it.end` dans de la mémoire appropriée (pile ou registres).

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Intervalles

- Les intervalles tels que `0..10` implémentent `IntoIterator` et `Iterator`.
- `for i in 0..N {...}` est facilement optimisé.
  - C'est en fait le moyen idiomatique d'écrire des boucles `for`.

Après l'inversion de `if` et `match`, et simplifications :

```
let mut it = Ran
loop {
  if it.start <
    let i = it.s
    it.start +=
    ...
  } else {
    break
  }
}
```

Ceci illustre le concept d'abstraction gratuite :

Le trait Iterator est une abstraction de la notion d'itérateur.

Mais, après la compilation, le code est aussi performant que si on avait fait l'itération à la main.

C'est très proche de ce qu'on peut attendre d'une boucle `for` en C/C++.
Il reste à :

- déplacer `it.start += 1` à la fin de la boucle ;
- placer `it.start` et `it.end` dans de la mémoire appropriée (pile ou registres).

Itérer sur Vec<T>

Il existe trois façons d'itérer sur un vecteur :

```
for x in v { ... }      // x:T,      propriété totale,     consomme v.
for x in &mut v { ... } // x:&mut T, emprunt unique de v,  modifie les éléments de v.
for x in &v { ... }     // x:&T,     emprunt partagé de v, laisse v tel quel.
```

Trois instances différentes pour `IntoIterator` (simplifiées) :

```
impl<T> IntoIterator for Vec<T> {
    type Item = T;
    type IntoIter = IntoIter<T>;
    fn into_iter(self) -> IntoIter<T> { ... }
}

impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    type IntoIter = Iter<'a, T>;
    fn into_iter(self) -> Iter<'a, T> { ... }
}

impl<'a, T> IntoIterator for &'a mut Vec<T> {
    type Item = &'a mut T;
    type IntoIter = IterMut<'a, T>;
    fn into_iter(self) -> IterMut<'a, T> { ... }
}
```

Itérer sur Vec<T>

Il existe trois façons d'itérer sur un vecteur :

```
for x in v { ... }      // x:T,      propriété totale,     consomme v.
for x in &mut v { ... } // x:&mut T, emprunt unique de v,  modifie les éléments de v.
for x in &v { ... }     // x:&T,     emprunt partagé de v, laisse v tel quel.
```

```
impl<'a, T> IntoIterator for &'a Vec<T> {
    type Item = &'a T;
    type IntoIter = Iter<'a, T>;
    fn into_iter(self) -> Iter<'a, T> { ... }
}
```

Note : dans le mode « emprunt », l'itérateur (e.g., `Iter<'a, T>`) contient un emprunt sur le vecteur !

$\Longrightarrow$ Il dépend de la durée de vie `'a`.

$\Longrightarrow$ `v` ne peut pas être utilisé tant que l'itérateur est vivant.

$\Longrightarrow$ Pas d'invalidation d'itérateur (cf premier cours) !

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Contrats de traits

Les traits ne sont pas qu'une liste de méthodes surchargées.
Ils correspondent aussi à des contrats que les implémentations doivent respecter.

Exemple :

```
// Dans std::clone
trait Clone {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

Le trait `Clone` a un contrat, explicitement décrit dans sa documentation :

- « `a.clone_from(&b)` est équivalent à `a = b.clone()`. »

Cette idée de contrat est très courante dans les traits définis en Rust.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Marker traits

Certains traits n'ont même pas de méthode. Ils ne font qu'exprimer un contrat.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Marker traits

Certains traits n'ont même pas de méthode. Ils ne font qu'exprimer un contrat.

C'est le cas de `Copy` : une instance `T: Copy` signifie que le type `T` peut être copié bit-à-bit. Exemples : `i32: Copy`, `u64: Copy`, `&T: Copy`...

Le compilateur sait désactiver le suivi de l'ownership lorsque `T: Copy`.

Lorsqu'on crée une instance de `Copy`, le compilateur vérifie qu'une telle copie de viole pas les règles d'ownership :

- Pour qu'un type `struct` puisse être `Copy`, il faut que tous ses champs le soient.
- Pour qu'un type `enum` puisse être `Copy`, il faut que ses constructeurs ne contiennent que des objets `Copy`.

Surcharge, traits et type classes
Programmation avancée
Jacques-Henri Jourdan
Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Marker traits

Certains traits n'ont même pas de méthode. Ils ne font qu'exprimer un contrat.

Le trait `Sized` exprime que la taille des objets d'un type est connue statiquement.

Pour manipuler directement (dans une variable locale) un objet de type `T`, il faut `T: Sized`. Sinon, on doit utiliser un pointeur vers `T` (`Box<T>`, `&T`, `&mut T`, ...).

Il est implémenté automatiquement par le compilateur.

Tous les types que nous avons vus jusqu'ici sont `Sized`.
Exemple de type qui n'est pas `Sized` : « slices » `[T]` (tableau dont la longueur n'est pas connue statiquement).

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Héritage

Certains traits sont des extensions d'autres traits.

Par exemple, le trait `Copy` hérite du trait `Clone` :

```
trait Copy: Clone {
  /* Rappel: Copy est un marker trait, il n'a pas de méthode. */
}
```

I.e., pouvoir copier bit-à-bit (`Copy`) donne une façon simple de cloner.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Héritage

Certains traits sont des extensions d'autres traits.

Par exemple, le trait `Copy` hérite du trait `Clone` :

```
trait Copy: Clone {
  /* Rappel: Copy est un marker trait, il n'a pas de méthode. */
}
```

I.e., pouvoir copier bit-à-bit (`Copy`) donne une façon simple de cloner.

Conséquence : quand on a une contrainte `T: Copy`, on peut aussi utiliser `clone` :

```
fn f<T: Copy>(x: T) { ... x.clone() ... }
```


Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Héritage

Ce qui se passe

En fait, quand on déclare qu'un trait hérite d'un autre :

```
trait B { ... }


trait A: B { ... }
```

```
trait B { ... }

// notation équivalente
trait A where Self: B { ... }
```

Il se passe deux choses :

- Le compilateur vérifie la contrainte `Self: B` pour chaque instance de `A`.
- Le compilateur crée une instance implicite `impl<T: A> B for T` qui peut être utilisée si on sait seulement que `T: A`.

Exemple d'héritage : Eq: PartialEq

Les opérateurs == et != sont surchargés par le trait `PartialEq`.

- À cause des nombres flottants, `PartialEq` n'a pas pour contrat que l'égalité soit une relation d'équivalence.
  - (== n'est pas toujours réflexif pour les flottants. On a `NaN != NaN`.)
- Il existe un marker trait `Eq` qui impose ce contrat.
  - Utile pour les structures de données qui en ont besoin (tables de hachage, ...)

```
trait PartialEq<Rhs> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... /* Impl par défaut */ }
}

trait Eq: PartialEq<Self> { }
```

Exemple d'héritage : Eq: PartialEq

Les opérateurs == et != sont surchargés par le trait `PartialEq`.

- À cause des nombres flottants, `PartialEq` n'a pas pour contrat que l'égalité soit une relation d'équivalence.
  - (== n'est pas toujours réflexif pour les flottants. On a `NaN != NaN`.)
- Il existe un marker trait `Eq` qui impose ce contrat.
  - Utile pour les structures de données qui en ont besoin (tables de hachage, ...)

```
trait PartialEq<Rhs> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { ... /* Impl par défaut */ }
}

trait Eq: PartialEq<Self> { }
```

Note : `PartialEq` peut être hétérogène, mais `Eq` impose `Rhs = Self`.
En effet, `Eq` ne prend pas `Rhs` en paramètre, et hérite de `PartialEq<Self>`
(et non de `PartialEq<Rhs>`).

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Exemple d'héritage

La hiérarchie PartialOrd/PartialEq/Ord/Eq

Les opérateurs de comparaison < > <= >= sont surchargés par le trait `PartialOrd`.
`PartialOrd` hérite de `PartialEq` (si on peut ordonner, on sait si deux objets sont égaux).

Comme pour `PartialEq`, `PartialOrd` n'est pas toujours une relation d'ordre totale.

Il existe un trait `Ord` pour les relations d'ordre totales.
`Ord` hérite à la fois de `Eq` et de `PartialOrd` !

Pour résumer :

```
trait PartialEq<Rhs> { ... }
trait Eq: PartialEq<Self> { }
trait PartialOrd<Rhs>: PartialEq<Rhs> { ... }
trait Ord: Eq + PartialOrd<Self> { ... }
```

Exemple d'héritage

La hiérarchie PartialOrd/PartialEq/Ord/Eq

Les opérateurs de comparaison < > <= >= sont surchargés par le trait `PartialOrd`.
`PartialOrd` hérite de `PartialEq` (si on peut ordonner, on sait si deux objets sont égaux).

Comme pour `PartialEq`, `PartialOrd` n'est pas toujours une relation d'ordre totale.

Il existe un trait `Ord` pour les relations d'ordre totales.
`Ord` hérite à la fois de `Eq` et de `PartialOrd` !

Pour résumer :

```
trait PartialEq<Rhs> { ... }
trait Eq: PartialEq<Self> { }
trait PartialOrd<Rhs>: PartialEq<Rhs> { ... }
trait Ord: Eq + PartialOrd<Self> { ... }
```

(Ouf !)

En cas d'ambiguïté ?

Que se passe-t-il si plusieurs instances conviennent pour un appel ?

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

En cas d'ambiguïté ?

Que se passe-t-il si plusieurs instances conviennent pour un appel ?

- Choisir une instance arbitraire ?
  - Mauvais choix : peu prévisible.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

En cas d'ambiguïté ?

Que se passe-t-il si plusieurs instances conviennent pour un appel ?

- Choisir une instance arbitraire ?
  - Mauvais choix : peu prévisible.
- Choisir l'instance la « plus spécifique » ?
  - Elle n'existe pas toujours.
  - Les versions récentes de Rust le font dans certains cas, mais c'est compliqué. (C'est la spécialisation.)

En cas d'ambiguïté ?

Que se passe-t-il si plusieurs instances conviennent pour un appel ?

- Choisir une instance arbitraire ?
  - Mauvais choix : peu prévisible.
- Choisir l'instance la « plus spécifique » ?
  - Elle n'existe pas toujours.
  - Les versions récentes de Rust le font dans certains cas, mais c'est compliqué. (C'est la spécialisation.)
- Faire une erreur ?
  - On vérifie donc que le programme est non-ambigu.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

En cas d'ambiguïté ?

Que se passe-t-il si plusieurs instances conviennent pour un appel ?

- Choisir une instance arbitraire ?
  - Mauvais choix : peu prévisible.
- Choisir l'instance la « plus spécifique » ?
  - Elle n'existe pas toujours.
  - Les versions récentes de Rust le font dans certains cas, mais c'est compliqué. (C'est la spécialisation.)
- Faire une erreur ?
  - On vérifie donc que le programme est non-ambigu.
- S'assurer que cela n'arrive jamais ?
  - C'est ce qu'on appelle la cohérence : on garantit qu'il ne peut exister qu'une seule instance concrète pour un trait donné (en fixant `Self` et les paramètres de trait).
  - On lève une erreur à la déclaration de l'instance qui crée l'incohérence.
  - C'est ce que vérifie Rust.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Pourquoi la cohérence ?

Le problème du diamant

En présence de polymorphisme et d'héritage, les ambiguïtés sont parfois inévitables.

Exemple :

```
trait A {
    fn a(self) {}
}
trait B: A {}
trait C: A {}

fn f<T: B + C>(x: T) {
   a(x)
}
```

Par héritage, l'instance `T: B` contient une instance `T: A`. Idem pour `T: C`.

Pour l'appel à `a`, doit-on prendre l'instance provenant de `B` ou de `C` ? Ambiguïté !

Avec la cohérence, pas de problème : ce ne peut être que la même instance de `T: A`.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Pourquoi la cohérence ?

Le problème de la table de hachage

Rappel :

```
impl<K: Eq + Hash, V> HashMap<K, V> {
....
    fn insert(&mut self, k: K, v: V) -> Option<V> { ... }
....
}
```

Il est crucial qu'une table de hachage utilise toujours la même fonction de hachage.
Il faut donc une garantie que l'instance `K: Hash` soit toujours la même.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Pourquoi la cohérence ?

Le problème de la table de hachage

Rappel :

```
impl<K: Eq + Hash, V> HashMap<K, V> {
....
    fn insert(&mut self, k: K, v: V) -> Option<V> { ... }
....
}
```

Il est crucial qu'une table de hachage utilise toujours la même fonction de hachage.
Il faut donc une garantie que l'instance `K: Hash` soit toujours la même.

Mais la non-ambiguïté ne suffit pas :

- `insert` peut être appelée dans des crates où seule une instance est visible.
- La table peut être partagée entre crates utilisant des instances différentes.

(Une *crate* est une unité de compilation en Rust.)

Par contre, la cohérence garantit qu'il n'existe au plus qu'une seule instance !

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Cohérence, comment ?

Au sein d'une même crate, on vérifie (par unification) qu'aucune paire d'instances est conflictuelle.

Mais comment se prémunir des problèmes de cohérence inter-crates ?

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Cohérence, comment ?

Au sein d'une même crate, on vérifie (par unification) qu'aucune paire d'instances est conflictuelle.

Mais comment se prémunir des problèmes de cohérence inter-crates ?

Avec les règles d'orphelinat (*orphan rules*) :

- Ensemble de règles assez techniques qui limitent la création d'instances pour garantir la cohérence en présence de plusieurs crates.
- Moralement, ce qu'il faut retenir (version simplifiée) :

Une instance est autorisée si soit le trait soit le type est définit dans le crate courant.

Ainsi, si deux instances sont conflictuelles, elles sont nécessairement dans la même crate ou dans deux crates qui dépendent l'une de l'autre.
$\Longrightarrow$ conflit facile à détecter par unification.

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

1 Introduction

2 Surcharge en Java

3 Type traits
- Intro : généricité et surcharge
- Premiers exemples
- Surcharge d'opérateurs
- Itérateurs
- Contrats de traits
- Héritage
- Cohérence

4 Typeclasses de Haskell

Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Type classes en Haskell

Haskell a une notion proche de la notion de trait : les type classes.

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool = [...]
  [...]
```

```
instance Ord a => Ord [a]
  compare x y = [...]

sort :: Ord t => [t] -> [t]
sort l = [...]
```

Première différence : mode de compilation :

- En Rust, les fonctions génériques sont monomorphisées.
  - I.e., les fonctions génériques sont dupliquées en autant de versions qu'il existe de types pour les instancier.
  - On connaît statiquement les instances des traits utilisés.
  - Tous les appels de fonctions surchargées sont compilés en appels directs.
- En Haskell, une fonction polymorphe n'est compilée qu'une fois.
  - Les instances de traits sont, à l'exécution, des dictionnaires.
  - Contraintes de type classes : dictionnaires passés en paramètres supplémentaires.
  - Les dictionnaires contiennent des pointeurs vers le code des fonctions surchargées.
  - Les appels de fonctions surchargées peuvent être indirects.

Surcharge, traits et type classes
Programmation avancée
Jacques-Henri Jourdan
Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell

Type classes en Haskell

Haskell a une notion proche de la notion de trait : les type classes.

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool = [...]
  [...]
```

```
instance Ord a => Ord [a]
  compare x y = [...]

sort :: Ord t => [t] -> [t]
sort l = [...]
```

Deuxième différence : expressivité

- Haskell supporte les paramètres de typeclasses de sorte supérieure.
- Exemple typique : `Monad M`.
  - `M` n'est pas un type. C'est un « constructeur de types » (ex : `Option`, `[.]`, ...).
- Ceci n'est pas possible en Rust, mais crucial en Haskell.

Surcharge, traits et type classes
Programmation avancée
Jacques-Henri Jourdan
Introduction
Surcharge en Java
Type traits
Intro : généricité et surcharge
Premiers exemples
Surcharge d'opérateurs
Itérateurs
Contrats de traits
Héritage
Cohérence
Typeclasses de Haskell