Fonctions comme valeur de première classe

Programmation avancée

Jacques-Henri Jourdan
(Cours en partie copié sur le cours de Jean-Christophe Filliâtre.)

12 avril 2023

Les fonctions imbriquées

Dans beaucoup de langages de programmation, on peut écrire des fonctions imbriquées :

```
let sum n =
  let f x = x * x in
  let rec loop i =
    if i = n then 0 else f i + loop (i+1)
  in
  loop 0
```

Le contenu de la fonction `loop` fait référence à `n` et `f`.

Les fonctions imbriquées

Les fonctions imbriquées existent depuis longtemps dans certains langages : Algol, Pascal, Ada, etc.

Leur compilation exploite le fait que toute variable qui est référencée est contenue dans un tableau d'activation quelque part sur la pile.

Le compilateur chaîne les tableaux d'activation, de façon à toujours pouvoir retrouver celui qui nous intéresse.

Fonction imbriquée

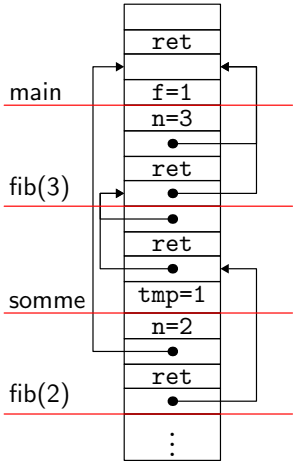
Exemple (en Pascal)

```
PROGRAM fib;

VAR f: INTEGER;

PROCEDURE fib(n: INTEGER);
  PROCEDURE somme();
  VAR tmp : INTEGER;
  BEGIN
    fib(n-2); tmp := f;
    fib(n-1); f := f + tmp
  END;
BEGIN
  IF n <= 1 THEN f := n ELSE somme()
END;

BEGIN fib(3); writeln(f) END
```



Ordre supérieur

Mais les fonctions imbriquées ne peuvent pas être passées en paramètre, ni être renvoyées.

OCaml le permet :

```
(* Cette fonction prend une fonction en paramètre. *)
let apply2 f x = f (f x)

(* Celle-ci en renvoie une. *)
let f x =
  if x < 0 then fun y -> y - x else fun y -> y + x
```

Dans ce dernier cas, la valeur renvoyée par `f` est une fonction qui utilise `x` mais le tableau d'activation de `f` vient précisément de disparaître !

$\Longrightarrow$ On ne peut donc pas compiler comme on le fait pour Pascal.

Ordre supérieur

Mais les fonctions imbriquées ne peuvent pas être passées en paramètre, ni être renvoyées.

OCaml le perme

```
(* Cette fonctio
let apply2 f x =

(* Celle-ci en r
let f x =
  if x < 0 then
```

Il existe deux manières de compiler les fonctions comme valeurs de premières classe.

- La closure conversion (et ses variantes) est l'objet du présent cours.
- La défonctionnalisation.

Dans ce dernier cas, la valeur renvoyée par `f` est une fonction qui utilise `x` mais le tableau d'activation de `f` vient précisément de disparaître !

$\Longrightarrow$ On ne peut donc pas compiler comme on le fait pour Pascal.

Mini-fragment d'OCaml

Considérons un mini-fragment d'OCaml :

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{fun}\ x\ \texttt{->}\ e \\
& | & e\ e \\
& | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
\\
p & ::= & d\ \ldots\ d
\end{array}
$$

Idée de la traduction

Les valeurs de fonctions sont compilées vers des clôtures (ou fermetures).

C'est une structure de donnée, allouée sur le tas (pour survivre aux appels de fonctions) contenant :

- un pointeur vers le code de la fonction à appeler,
- les valeurs des variables susceptibles d'être utilisées par ce code : l'environnement de la clôture.

Chaque occurrence de `fun` dans le source donne lieu à exactement un symbole de fonction dans le code (assembleur) final.
Le comportement de ce bloc de code assembleur est modifié par le contenu de l'environnement qui lui est associé.

Variables de l'environnement

Quelles sont justement les variables qu'il faut mettre dans l'environnement de la clôture représentant `fun` x `->` e ?

Ce sont les variables libres de `fun` x `->` e.

I.e., ce sont les variables apparaissant dans ce terme, mais qui n'y sont pas liées :

$$
\begin{aligned}
fv(c) &= \emptyset \\
fv(x) &= \{x\} \\
fv(\texttt{fun}\ x\ \texttt{->}\ e) &= fv(e) \setminus \{x\} \\
fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) &= fv(e_1) \cup (fv(e_2) \setminus \{x\}) \\
fv(\texttt{let rec}\ x = e_1\ \texttt{in}\ e_2) &= (fv(e_1) \cup fv(e_2)) \setminus \{x\} \\
fv(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= fv(e_1) \cup fv(e_2) \cup fv(e_3)
\end{aligned}
$$

Exemple

Considérons le programme suivant qui approxime $\int_0^1 x^n dx$:

```
let rec pow i x =
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum x =
    if x >= 1. then 0.
    else f x +. sum (x +. eps)
  in
  sum 0. *. eps
```

Exemple

Considérons le programme suivant qui approxime $\int_0^1 x^n dx$:

```
let rec pow = fun i -> fun x ->
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum = fun x ->
    if x >= 1. then 0.
    else f x +. sum (x +. eps)
  in
  sum 0. *. eps
```

Commençons par déplier le sucre syntaxique.

Exemple

Considérons le programme suivant qui approxime $\int_0^1 x^n dx$:

```
let rec pow = fun i -> fun x ->
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum = fun x ->
    if x >= 1. then 0.
    else f x +. sum (x +. eps)
  in
  sum 0. *. eps
```

Pour cette clôture, l'environnement est {`pow`}.

Exemple

Considérons le programme suivant qui approxime $\int_0^1 x^n dx$:

```
let rec pow = fun i -> fun x ->
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum = fun x ->
    if x >= 1. then 0.
    else f x +. sum (x +. eps)
  in
  sum 0. *. eps
```

Pour cette clôture, l'environnement est $\{\texttt{pow}, \texttt{i}\}$.

Exemple

Considérons le programme suivant qui approxime $\int_0^1 x^n dx$:

```
let rec pow = fun i -> fun x ->
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum = fun x ->
    if x >= 1. then 0.
    else f x +. sum (x +. eps)
  in
  sum 0. *. eps
```

Pour celle-ci, l'environnement est {pow}.

Exemple

Considérons le programme suivant qui approxime $\int_0^1 x^n dx$:

```
let rec pow = fun i -> fun x ->
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum = fun x ->
    if x >= 1. then 0.
    else f x +. sum (x +. eps)
  in
  sum 0. *. eps
```

Et pour celle-ci, l'environnement est $\{\texttt{eps}, \texttt{f}, \texttt{sum}\}$.

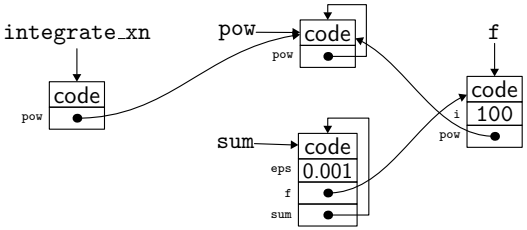
Exemple

Considérons le programme suivant qui approxime $\int_0^1 x^n dx$:

```
let rec pow = fun i -> fun x ->
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum = fun x ->
    if x >= 1. then 0.
    else f x +. sum (x +. eps)
  in
  sum 0. *. eps
```

Pendant l'exécution de `integrate_xn 100`, on a quatre clôtures :



Closure conversion

Pour compiler les fonctions comme valeurs de première classe, on peut procéder en deux temps :

1. On recherche dans le code toutes les constructions `fun` x -> e, et on les remplace par une opération explicite de construction de clôture :

$$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

où les y_i sont les variables libres de `fun` x -> e, et f le nom (généré) donné à une nouvelle déclaration globale de la fonction de la forme :

$$\texttt{letfun}\ f\ [y_1, \ldots, y_n]\ x = e'$$

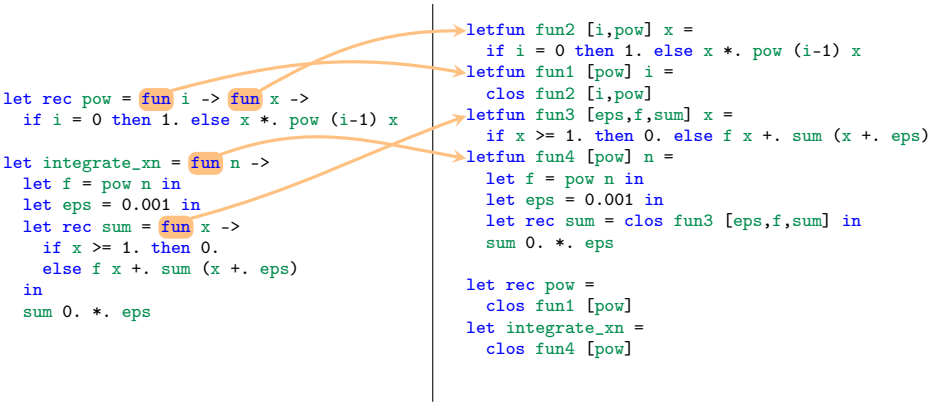
C'est la closure conversion.

2. On poursuit la compilation du code obtenu, qui ne contient plus de fonction comme valeur de première classe.

Closure conversion

Exemple

Reprenons notre exemple :



Suite de la compilation

`letfun`

Chaque symbole de fonction introduit par `letfun` peut être compilé indépendamment des autres, (comme une fonction en C, par exemple). Ces fonctions ont deux paramètres :

- Le paramètre de la fonction initiale (le x dans `fun` x `->` e).
- Un pointeur vers la clôture, contenant l'environnement.

Dans le corps de la fonction, les valeurs des variables libres sont à chercher dans l'environnement de la clôture passée en paramètre.

Suite de la compilation

Application

Pour compiler e_1 e_2 on génère du code qui évalue les opérations suivantes.

- Évalue e_2 et e_1. Elle renvoie les valeurs v_2 et v_1.
  Par typage, on sait que v_1 est une clôture.
- On appelle le pointeur de code de la clôture v_1, en passant en paramètres :
  - v_2 (le paramètre initial),
  - v_1 (la clôture, pour qu'elle puisse accéder à son environnement).

Suite de la compilation

`clos`

L'opération `clos` $f\ [y_1, \ldots, y_n]$ est compilée vers du code qui :

- alloue un bloc de taille $n + 1$;
- écrit un pointeur de code vers f dans le premier champ du bloc ;
- écrit $y_1, \ldots, y_n$ dans les champs suivants ;
- renvoie un pointeur vers le bloc.

Optimisations

En pratique, le compilateur `ocamlopt` effectue quelques optimisations pour obtenir des performances raisonnables :

- Les variables (dont les fonctions) globales sont accédées directement, sans passer par l'environnement de la clôture en cours d'exécution.
- Lorsque l'on appelle une clôture dont on connaît le code (cas le plus fréquent), on fait un appel direct sans passer par le pointeur de code.
- Fonctions curryfiées : un mécanisme évite la construction des clôtures intermédiaires si le nombre de paramètres attendus correspond au nombre de paramètres fournis.

En particulier, la closure conversion n'a pas d'impact en performances pour l'appel direct d'une fonction globale à plusieurs paramètres.

Clôtures en Rust : introduction

La notion de type de fonction (e.g., τ_1 -> τ_2 en OCaml) est trop dynamique.

- Taille de l'environnement non connu statiquement $\Longrightarrow$ il faut le manipuler à travers un pointeur (sur le tas).
- Utilisation d'un pointeur de code $\Longrightarrow$ appels plus lents, optimisation difficile.
- $\Longrightarrow$ En Rust, les fonctions sont un trait, pas un type.

Il faut prendre en compte les ressources des variables capturées.

- La clôture crée en fait un alias des variables capturées.
- Appeler la clôture la consomme-t-elle ? Peut-on appeler la clôture de manière réentrante ?
- $\Longrightarrow$ Il y a trois traits pour les clôtures : `Fn`, `FnMut` et `FnOnce`.

Le trait FnOnce

```
trait FnOnce<Args> {
      type Output;
      fn call_once(self, args: Args) -> Self::Output;
}
```

`Args` est un type de *n*-uplet pour les paramètres de la clôture.

Note : syntaxe spéciale pour ce trait.
On écrit « `FnOnce(T1, ..., Tn) -> R` » pour « `FnOnce<(T1, ..., Tn), Output = R>` ».

Le trait FnOnce

```
trait FnOnce<Args> {
      type Output;
      fn call_once(self, args: Args) -> Self::Output;
}
```

`Args` est un type de *n*-uplet pour les paramètres de la clôture.

Note : syntaxe spéciale pour ce trait.
On écrit « `FnOnce(T1, ..., Tn) -> R` » pour « `FnOnce<(T1, ..., Tn), Output = R>` ».

Lorsque l'utilisateur écrit la fonction `|a: &mut u64, b: i32| ...`, Rust génère au moment du typage :

- un type `struct` anonyme : l'environnement (chaque variable capturée : un champ),
- une instance de `FnOnce<(&mut u64, i32)>` pour le type de l'environnement,
  - `Output` : le type de retour de la fonction,
  - `call_once` : le code de la fonction (variables capturées : via `self`).

Le trait FnOnce

```
trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}
```

Args est un typ

Note : syntaxe s
On écrit « FnOn

Lorsque l'utilisa
moment du typ

- un type st
- une instance de `FnOnce<(&mut u64, i32)>` pour le type de l'environnement,
  - `Output` : le type de retour de la fonction,
  - `call_once` : le code de la fonction (variables capturées : via `self`).

Ceci est une forme de closure conversion, mais pendant le typage.

Chaque clôture a un type différent, mais elles implémentent toutes `FnOnce`.

C'est une stratégie similaire qui est utilisée en Java ou en C++.

Le trait FnOnce et le suivi de la possession

```
trait FnOnce<Args> {
      type Output;
      fn call_once(self, args: Args) -> Self::Output;
}
```

La méthode `call_once` récupère la possession totale de `self`.

- Le corps de la clôture peut avoir la possession totale des variables capturées.
- La clôture ne peut être appelée qu'une seule fois.

(On verra dans un instant les traits `FnMut` et `Fn` qui fournissent un autre protocole.)

Exemple d'utilisation de `FnOnce` : `once_with`

`std::iter::once_with(f)` renvoie un itérateur qui :

- appelle la clôture `f` lors du premier appel à `next`, et renvoie le résultat ;
- renvoie `None` lors de tous les autres appels à `next`.

Exemple d'utilisation de FnOnce : once_with

`std::iter:once_with(f)` renvoie un itérateur qui :

- appelle la clôture `f` lors du premier appel à `next`, et renvoie le résultat ;
- renvoie `None` lors de tous les autres appels à `next`.

```
fn once_with<A, F: FnOnce() -> A>(gen: F) -> OnceWith<F> {
    OnceWith { gen: Some(gen) }
}

struct OnceWith<F> { gen: Option<F> }

impl<A, F: FnOnce() -> A> Iterator for OnceWith<F> {
    type Item = A;
    fn next(&mut self) -> Option<A> {
        match std::mem::replace(&mut self.gen, None) {
            None => None
            Some(f) => Some(f())
        }
    }
}
```

Exemple d'utilisation de FnOnce : once_with

`std::iter:once_with(f)` renvoie un itérateur qui :

- appelle la clôture `f` lors du premier appel à `next`, et renvoie le résultat ;
- renvoie `None` lors de tous les autres appels à `next`.

```
fn once_with<A, F: FnOnce() -> A>(gen: F) -> OnceWith<F> {
    OnceWith { gen: Some(gen) }
}

struct OnceWith<F> { gen: Option<F> }

impl<A, F: FnOnce() -> A> Iterator for OnceWith<F> {
    type Item = A;
    fn next(&mut self) -> Option<A> {
        match std::mem::replace(&mut self.gen, None) {
            None => None
            Some(f) => Some(f())
        }
    }
}
```

Sucre syntaxique pour `f.call_once(())`.
(Appel de la clôture.)

Exemple d'utilisation de FnOnce : once_with

`std::iter:once_with(f)` renvoie un itérateur qui :

- appelle la clôture `f` lors du premier appel à `next`, et renvoie le résultat ;
- renvoie `None` lors de tous les autres appels à `next`.

```
fn once_with<A, F: FnOnce() -> A>(gen: F) -> OnceWith<F> {
    OnceWith { gen: Some(gen) }
}

struct OnceWith<F> { gen: Option<F> }

impl<A, F: FnOnce() -> A> Iterator for OnceWith<F> {
    type Item = A;
    fn next(&mut self) -> Option<A> {
        match std::mem::replace(&mut self.gen, None) {
            None => None
            Some(f) => Some(f())
        }
    }
}
```

Tout est paramétré par `F`, le type de la clôture.

La contrainte `F: FnOnce() -> A` impose que c'est une clôture, avec les bons types de retour et de paramètre.

Le code de l'itérateur est compilé une fois pour chaque clôture.
Avantage : appel direct, optimisations (inlining).
Inconvénient : duplication du code.

Exemple d'utilisation de FnOnce : once_with

Création d'une clôture

```
let mut x: Box<i32> = Box::new(42);
let y: Box<i32> = Box::new(12);

for u in once_with(|| { *x += *y; x }) {
    println!("{}", *u)
}

// ERREUR, ‘x‘ est consommé dans la clôture.
println!("{}", *x);

// OK, ‘y‘ a seulement été emprunté
println!("{}", *y);
```

Traduit vers :

```
struct F<'a> {
    x: Box<i32>,
    y: &'a Box<i32>
}
impl<'a> FnOnce<()> for F<'a> {
    type Output = Box<i32>;
    fn call_once(self, _ : ()) -> Box<i32> {
        *self.x += *(*self.y); self.x
    }
}

let mut x: Box<i32> = Box::new(42);
let y: Box<i32> = Box::new(12);

let f = F { x: x, y: &y };
for u in once_with(f) {
   ...
```

D'autres genres de clôtures

`FnOnce`, `FnMut`, `Fn`

Que faire si on veut appeler une clôture plusieurs fois ?

Ou même, appeler une clôture de manière réentrante (i.e., plusieurs fois en même temps) ?

D'autres genres de clôtures

FnOnce, FnMut, Fn

Que faire si on veut appeler une clôture plusieurs fois ?

Ou même, appeler une clôture de manière réentrante (i.e., plusieurs fois en même temps) ?

Il existe en fait trois traits de clôtures en Rust :

```
trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

trait FnMut<Args>: FnOnce<Args> { // Héritage
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

trait Fn<Args>: FnMut<Args> { // Héritage
    fn call(&self, args: Args) -> Self::Output;
}
```

D'autres genres de clôtures

FnOnce, FnMut, Fn

Que faire si on veut appeler une clôture plusieurs fois ?

Ou même, appeler une clôture de manière réentrante (i.e., plusieurs fois en même temps) ?

Il existe en fait trois traits de clôtures en Rust :

```
trait FnOnce<Args> {
    type Output;
    fn call_once(self, args: Args) -> Self::Output;
}

trait FnMut<Args>: FnOnce<Args> { // Héritage
    fn call_mut(&mut self, args: Args) -> Self::Output;
}

trait Fn<Args>: FnMut<Args> { // Héritage
    fn call(&self, args: Args) -> Self::Output;
}
```

Quand on écrit une clôture, le compilateur génère le plus d'instances possibles, en fonction du type d'utilisation des variables capturées (transfert de possession, emprunt unique ou partagé).

API avec clôtures

Combinateurs d'itérateur

Il existe de nombreux combinateurs d'itérateur.
Exemples :

```
trait Iterator {
...
  fn map<B, F>(self, f: F) -> Map<Self, F>
    where F: FnMut(Self::Item) -> B { ... }
  fn filter<P>(self, predicate: P) -> Filter<Self, P>
    where P: FnMut(&Self::Item) -> bool { ... }
...
}

impl<B, I: Iterator, F> Iterator for Map<I, F>
where F: FnMut(I::Item) -> B,
{ type Item = B; ... }

impl<I: Iterator, P> Iterator for Filter<I, P>
where P: FnMut(&I::Item) -> bool,
{ type Item = I::Item; ... }
```

On peut par exemple écrire :

```
(0..N).filter(|i| *i % 2 == 0).map(|i| i * i).sum()
```

Abstraction gratuite avec des itérateurs et des clôtures

Regardons comment le code suivant est compilé :

```
(0..N).filter(|i| *i % 2 == 0).map(|i| i * i + a).sum()
```

On commence par la closure conversion...

Abstraction gratuite avec des itérateurs et des clôtures

Après la closure conversion :

```
struct C1 { /* Pas de variable capturée */ }
impl FnOnce<(&i32,)> for C1 { type Output = i32; ... }
impl FnMut<(&i32,)> for C1 {
  fn call_mut(&mut self, args:(&i32,)) -> bool { *args.0 % 2 == 0 }
}

struct C2 { a: i32 }
impl FnOnce<(i32,)> for C2 { type Output = i32; ... }
impl FnMut<(i32,)> for C2 {
  fn call_mut(&mut self, args:(i32,)) -> i32 { args.0 * args.0 + self.a }
}

(0..N).filter(C1 {}).map(C2 { a: a }).sum()
```

Maintenant, on *inline* `filter`, `map` et `sum`...
(I.e., on déplie leur définition.)

Abstraction gratuite avec des itérateurs et des clôtures

After l'*inlinling* de `filter`, `map` et `sum` :

```
/* Définition et instances de C1 et C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter: Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  match it.next() {
    None => break,
    Some(i) => r += i
  }
}
```

On *inline* `<Map>::next`...

Abstraction gratuite avec des itérateurs et des clôtures

Après l'*inlining* de `<Map>::next` :

```
/* Définition et instances de C1 et C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter: Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  let o = match it.iter.next() { None => None, Some(i) => Some(it.f.call_mut(i)) };
  match o {
    None => break,
    Some(i) => r += i
  }
}
```

Important : `call_mut` est un appel direct !

On peut fusionner les deux constructions `match`, et *inliner* `<C2>::call_mut`...

Abstraction gratuite avec des itérateurs et des clôtures

Après l'*inlining* de `<C2>::call_mut` et quelques simplifications :

```
/* Définition et instances de C1 et C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter: Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  match it.iter.next() {
    None => break,
    Some(i) => r += i*i+it.f.a
  }
}
```

De manières similaire, on peut *inliner* `<Filter>::next` et `<C1>::call_mut`...

Abstraction gratuite avec des itérateurs et des clôtures

Après l'*inlining* de `<Filter>::next` et `<C1>::call_mut` :

```
/* Définition et instances de C1 et C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter: Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  let o;
  loop { match it.iter.iter.next() {
    None => { o = None; break },
    Some(i) => if i % 2 == 0 { o = Some(i); break }
  } }
  match o {
    None => break,
    Some(i) => r += i*i+it.f.a
  }
}
```

Le compilateur opère sur un *graphe de flot de contrôle*.
Dans cette représentation, il est facile de voir que les deux `match` peuvent être fusionnées...

Abstraction gratuite avec des itérateurs et des clôtures

On a fusionné les deux `match`, et simplifié :

```
/* Définition et instances de C1 et C2. */
...

let mut it = Map {
  f: C2 { a: a },
  iter: Filter {
      predicate: C1 {},
      iter: Range { start: 0, end: N }
    }
};
let mut r = 0;
loop {
  match it.iter.iter.next() {
    None => break,
    Some(i) => if i % 2 == 0 { r += i*i+it.f.a }
  }
}
```

On *inline* `<Range>::next()` et on déplie les parties de `it` qui ne changent pas...

Abstraction gratuite avec des itérateurs et des clôtures

... et on obtient finalement :

```
let mut range_start = 0;
loop {
  if range_start < N {
    let i = range_start;
    range_start += 1;
    if i % 2 == 0 { r += i*i+a }
  } else { break }
}
```

On est parti des ce code concis et « abstrait » :

```
(0..N).filter(|i| *i % 2 == 0).map(|i| i * i + a).sum()
```

Et on termine avec une version bien optimisée !
Aucune étape d'optimisation n'était réellement difficile. Beaucoup de compilateurs moderne en sont capables.

Un exemple d'abstraction gratuite !

Renvoyer une fonction

En OCaml, on peut écrire

```
let f x =
  if x.(0) < 0 then fun y -> y - x.(1) else fun y -> y + x.(1)
```

I.e., en OCaml, on peut écrire une fonction qui renvoie une clôture.

En Rust, il y a plusieurs problèmes pour écrire cette fonction. Lesquels ?

Renvoyer une fonction

En OCaml, on peut écrire

```
let f x =
  if x.(0) < 0 then fun y -> y - x.(1) else fun y -> y + x.(1)
```

I.e., en OCaml, on peut écrire une fonction qui renvoie une clôture.

Quand on écrit une clôture, son type est anonyme.
On ne peut donc pas écrire le type de retour de notre fonction :

```
fn f() -> ???? {
    |y: i32| { y }
}
```

Renvoyer une fonction

En OCaml, on peut écrire

```
let f x =
  if x.(0) < 0 then fun y -> y - x.(1) else fun y -> y + x.(1)
```

I.e., en OCaml, on peut écrire une fonction qui renvoie une clôture.

Mais on peut écrire :

```
fn f() -> impl Fn(i32) -> i32 {
    |y: i32| { y }
}
```

La syntaxe « `impl Fn(i32) -> i32` » signifie :
« Un type implicite, mais que le compilateur devinera. Il implémentera `Fn(i32) -> i32`. »

Renvoyer une fonction

En OCaml, on peut écrire

```
let f x =
  if x.(0) < 0 then fun y -> y - x.(1) else fun y -> y + x.(1)
```

I.e., en OCaml, on peut écrire une fonction qui renvoie une clôture.

Par contre, on ne peut pas écrire :

```
fn f(x: Vec<i32>) -> impl Fn(i32) -> i32 {
    |y: i32| { y + x[1] }
}
```

Pourquoi ?

Renvoyer une fonction

En OCaml, on p

```
let f x =
  if x.(0) < 0 t
```

I.e., en OCaml,

Par contre, on r

```
fn f(x: Vec<i32>
    |y: i32| { y
}
```

Pourquoi ?

```
 --> test.rs:3:27
  |
3 |     |y: i32| { y + x[0] }
  |     --------       ^ borrowed value does not live long enough
  |     |
  |     value captured here
4 | }
  | -- borrow later used here
  | |
  | `x` dropped here while still borrowed
```

`x` est dans le tableau d'activation associé à `f`.
La clôture contient un emprunt sur `x` (puisqu'on ne s'en sert que de cette façon).
Si on renvoie la clôture en sortant de `f`, elle contient un emprunt sur `x`, qui n'existe plus !

Renvoyer une fonction

En OCaml, on peut écrire

```
let f x =
  if x.(0) < 0 then fun y -> y - x.(1) else fun y -> y + x.(1)
```

I.e., en OCaml, on peut écrire une fonction qui renvoie une clôture.

Pour résoudre ce problème, on peut forcer Rust à transférer la propriété de `x` dans la clôture, avec le mot-clé `move` :

```
fn f(x: Vec<i32>) -> impl Fn(i32) -> i32 {
    move |y: i32| { y + x[1] }
}
```

Renvoyer une fonction

En OCaml, on peut écrire

```
let f x =
  if x.(0) < 0 then fun y -> y - x.(1) else fun y -> y + x.(1)
```

I.e., en OCaml, on peut écrire une fonction qui renvoie une clôture.

Et maintenant, peut-on écrire l'équivalent du code OCaml ?

```
fn f(y: Vec<i32>) -> impl Fn(i32) -> i32 {
    if x[0] < 0 { move |y: i32| { y - x[1] } }
    else { move |y: i32| { y + x[1] } }
}
```

Alors, selon vous, est-ce accepté en Rust ?

Renvoyer une fonction

En OCaml, on

```
let f x =
  if x.(0) < 0
```

I.e., en OCaml

Et maintenant

```
fn f(y: Vec<i3
    if x[0] <
    else { mov
}
```

Alors, selon v

```
error[E0308]: ‘if‘ and ‘else‘ have incompatible types
 --> test.rs:3:12
  |
2 |       if x[0] < 0 { move |y: i32| { y - x[1] } }
  |       -             ------------------------- expected because of this
  |  _____|
  | |
3 | |     else { move |y: i32| { y + x[1] } }
  | |___________^^^^^^^^^^^^^^^^^^^^^^^^^_- ‘if‘ and ‘else‘ have incompatible types
  |             |
  |             expected closure, found a different closure
  |
  = note: expected type ‘[closure@test.rs:2:19: 2:45]‘
          found closure ‘[closure@test.rs:3:12: 3:38]‘
  = note: no two closures, even if identical, have the same type
  = help: consider boxing your closure and/or using it as a trait object
```

Souvenez-vous : chaque clôture a un type différent.
Le type de retour, `impl Fn(i32) -> i32`, signifie que le type de la clôture est implicite, mais connu du compilateur.
Les deux branches du `if` doivent avoir le même type, donc erreur !

Renvoyer une fonction

En OCaml, on peut écrire

```
let f x =
  if x.(0) < 0 then fun y -> y - x.(1) else fun y -> y + x.(1)
```

I.e., en OCaml, on peut écrire une fonction qui renvoie une clôture.

Et maintenant,

On fait comment ?

```
fn f(y: Vec<i32>) -> impl Fn(i32) -> i32 {
    if x[0] < 0 { move |y: i32| { y - x[1] } }
    else { move |y: i32| { y + x[1] } }
}
```

Alors, selon vous, est-ce accepté en Rust ?

Trait objects

De temps en temps, on a besoin d'un vrai type de fonction :

- Parce qu'on ne peut pas toujours spécialiser le code par le type des clôtures.
- Parce que trop de spécialisation augmente la taille du code généré.

Solution : certains traits peuvent être transformés en un type, appelé trait object.

- Syntaxe : `dyn Trait`
  - Dans notre cas, on écrirait `dyn Fn(i32) -> i32`.
- C'est une sorte de type existentiel.
  - Si on sait `v: dyn Trait`, alors il existe un type `T` tel que `v: T` et `T: Trait`.
- La taille des trait objects est inconnue statiquement (ils n'implémentent pas `Sized`).
  - Ils ne peuvent être manipulés qu'à travers des pointeurs : `Box<dyn Trait>`, `&dyn Trait`, ...

De retour à notre exemple

On voulait écrire l'équivalent en Rust de :

```
let f x =
  if x.(0) < 0 then fun y -> y - x.(1) else fun y -> y + x.(1)
```

On peut le faire avec des trait objects :

```
fn f(x: Vec<i32>) -> Box<dyn Fn(i32) -> i32> {
    if x[0] < 0 { Box::new(move |y: i32| { y - x[1] }) }
    else { Box::new(move |y: i32| { y + x[1] }) }
}
```

Et ça marche (mais c'est un peu plus compliqué qu'en OCaml) !

dyn Trait vs. impl Trait

Quelle sont les différences entre `fn f() -> Box<dyn Trait>` et `fn f() -> impl Trait` ?

`impl Trait` est un type implicite : le compilateur infère le type dont il s'agit.

- Donc il connaît sa taille (pas la peine d'utiliser un pointeur).
- Donc les appels de méthodes sont des appels directs.

`dyn Trait` est un type existentiel : le compilateur ne connaît pas statiquement le type sous-jacent.

- Donc on ne connaît pas la taille statiquement.
  Il faut utiliser un pointeur (`Box`, `&` ou `&mut`).
- Donc les appels de méthodes nécessitent une technique particulière.

Compilation des trait objects

Le problème

On a un trait (par exemple `Fn`, `FnMut` ou `FnOnce`, mais pas forcément), et des implémentations de ce trait :

```
trait Tr {
    fn f(&mut self) -> i32;
    fn g(&mut self) -> i32;
}

struct T1 {}
impl Tr for T1 {
  fn f(&mut self) -> i32 { 1 }
  fn g(&mut self, x: i32) -> i32 { x+1 }
}

struct T2 {}
impl Tr for T2 {
  fn f(&mut self) -> i32 { 2 }
  fn g(&mut self, x: i32) -> i32 { x+2 }
}
```

Si `x: Box<dyn Tr>`, comment fait-on pour compiler un appel `x.f()` ?
Rappel : on ne sait pas quel est le type de `*x`, donc on ne sait pas quelle méthode appeler.

Compilation des trait objects

Pointeurs lourds, vtables

Ce problème est le problème du dynamic dispatch.
Rust utilise une technique de compilation classique, qui vient de la compilation des appels de méthodes virtuelles dans langages orientés objets (ex. C++, Java...).
On va donc décrire la technique dans le cadre de Rust, mais elle s'applique aussi aux langages orientés objet.

Compilation des trait objects

Pointeurs lourds, vtables

Ce problème est le problème du dynamic dispatch.
Rust utilise une technique de compilation classique, qui vient de la compilation des appels de méthodes virtuelles dans langages orientés objets (ex. C++, Java...).
On va donc décrire la technique dans le cadre de Rust, mais elle s'applique aussi aux langages orientés objet.

Un pointeur `Box<dyn Trait>` n'est pas un pointeur classique (comme `Box<i32>`).
C'est un pointeur lourd.

Un pointeur lourd occupe deux mots en mémoire :

- Le premier est l'adresse de l'objet pointé (comme un pointeur classique).
- Le deuxième est un pointeur vers une vtable, ou table des méthodes virtuelles, spécifique au type.

Note : une autre possibilité, utilisée dans des langages orientés objets, et de stocker le pointeur vers la vtable directement en en-tête de l'objet (plutôt que dans un pointeur qui pointe vers l'objet).

Vtables

Une vtable est une structure de donnée construite par le compilateur.
Il en construit une pour chaque implémentation d'un trait pour un type.
C'est une table, avec un pointeur de code pour chaque méthode du trait, pour le type en question.

Ainsi, si `x: Box<dyn Tr>`, le code généré pour un appel `x.f()` :

- déréférence le pointeur de vtable contenu dans `x`, et lit la vtable ;
- trouve un pointeur vers le code approprié pour `f` dans la vtable ;
- effectue un appel indirect vers ce pointeur de code, en passant en premier paramètre le pointeur vers la valeur contenu dans `x`.

Vtables et héritage

Si un trait `A` hérite d'un trait `B`, on peut appeler une méthode de `B` à partir d'une méthode de `A`.
Ainsi, une vtable pour `A` contient aussi des pointeurs de code pour toutes les méthodes de `B`.

Vtables et héritage

Si un trait `A` hérite d'un trait `B`, on peut appeler une méthode de `B` à partir d'une méthode de `A`.
Ainsi, une vtable pour `A` contient aussi des pointeurs de code pour toutes les méthodes de `B`.

Par ailleurs, il est possible d'upcaster un `Box<dyn B>` vers un `Box<dyn A>`. (En Rust ceci nécessite d'activer certaines fonctionnalités non-stables. Mais c'est possible dans n'importe quel langage orienté objet.)
On obtient ce comportement en :

- s'arrangeant pour qu'une vtable pour `A` soit un préfixe d'une vtable pour `B`,
  - (Ainsi un pointeur vers une vtable pour `A` est aussi un pointeur vers une vtable pour `B`.)
- ou, lorsque cela n'est pas possible, en stockant un pointeur vers une vtable pour `B` dans la vtable pour `A`.
  - (Il faut alors transformer le pointeur vers la vtable de `B` en un pointeur vers la vtable de `A`, en allant chercher dans la table.)

Contraintes sur les trait objects

Le schéma de compilation par vtable n'est pas compatible avec tous les traits.
Certains traits ne peuvent donc pas être utilisés dans des traits objects.
Les traits autorisés sont appelés object safe.

Pour être object safe, un trait doit :

- ne pas avoir de méthode générique,
- ne pas utiliser `Self` dans le type de retour ou des arguments d'une méthode (sauf le premier paramètre),
- respecter d'autres contraintes techniques.