Programmation avancée

TD 08 : Clôtures et trait objects

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse Paul Robert

Pensez bien à vérifier que votre code compile. Si vous n'avez pas terminé, remplacez les parties manquantes par `todo!()`.

Pensez également à tester votre implémentation avec `cargo test`.

Note Téléchargez le code de démarrage de ce TP à l'adresse suivante : `https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/08-clotures/td08.zip`

1 Traits Objects

Fichier `rust/src/partie1.rs`

Exercice 1. *Compléter les fonctions associées de* `SomeVec`*, afin que le test* `multi_vec` *passe.*

Exercice 2. *Écrire la fonction* `fn last_elem_is_int(v: &SomeVec) -> bool`*, qui vérifie si le dernier élément du vecteur donné est un entier (*`i32`*).*

Exercice 3. *Écrire la fonction* `fn printable_vec(v: &[Box<dyn Debug>])`*, qui imprime le contenu d'un vecteur d'objets imprimables.*

Exercice 4. *Pourquoi ne peut on pas utiliser le trait* `Foo` *ci-dessous dans un trait object (créer un* `dyn Foo`*) ?*

```
trait Foo {
    type Assoc;

    fn into_assoc(self) -> Self::Assoc;
}
```

Justifier votre réponse en faisant référence au modèle de compilation de Rust.

Trouvez tout de même un moyen de faire compiler la fonction `fn fixme(b: bool) -> Box<dyn Foo>` *dans le code du TP,* ***en ne changeant que sa signature****.*

Exercice 5. *Pourquoi ne peut on pas écrire le code suivant:*

```
fn wont_compile<I : Iterator>(x : I, b: bool) -> impl Iterator {
    if b {
        x
    } else {
        x.skip(5)
    }
}
```

Expliquer, et corriger le code (Bien lire les messages du compilateur !).

2 Clôtures

Fichier `rust/src/partie2.rs`

En Rust, la syntaxe des clôtures est

```
let ma_cloture = |args| expr;
```

On peut:

- Omettre le type des arguments, ainsi que le type de retour ;
- Mettre un bloc contenant plusieurs instructions dans `expr` (un bloc est une expression !) ;
- Utiliser les variables environnantes ;
- Ajouter `move` devant la clôture pour capturer par valeur.

Exercice 6. *Corriger la fonction suivante, afin qu'elle affiche `2 5`, en ajoutant un seul mot-clef :*

```
fn affiche() {
    let mut x = 1;
    let f1 = || {
        x += 1;
        x
    };
    let f2 = || {
        x += 4;
        x
    };
    println!("{} {}", f1(), f2());
}
```

Exercice 7. *Une clôture peut-elle être `Copy` ? Si oui, donner un exemple, si non, expliquer.*

Le combinateur `map` nous permet d'appliquer une clôture à tout les elements d'un itérateur. Par exemple, `iter.map(|x| x)` applique la fonction d'identité à chaque élément.

Exercice 8. *Écrire la clôture qui permets d'incrémenter les valeurs de l'itérateur par* `1`.

```
#[test]
fn increment() {
    let iter = (0..10).map(todo!());

    assert!(Iterator::eq(iter, 1..11));
}
```

Exercice 9. *Écrire une fonction* `fn count_with_map<I : Iterator>(i : I) -> usize` *qui utilise* `map` *pour compter le nombre d'elements dans l'itérateur.*

Pour les questions suivantes nous allons utiliser ces trois fonctions:

```
fn call_fn_once<F : FnOnce()>(f : F) { f () }

fn call_fn_mut<F : FnMut()>(mut f : F) { f () }

fn call_fn<F : Fn()>(f : F) { f () }
```

Exercice 10. *Est-il possible d'écrire une clôture qui compile quand on la passe en argument à* `call_fn` *mais pas à* `call_fn_mut` *? Si oui, donner un exemple, si non, expliquer pourquoi pas.*

Exercice 11. *Pour chaque exemple, déterminer si* `cloture` *implémente* `Fn`, `FnMut` *et/ou* `FnOnce`.

1.
```
let x: T = ...;
let cloture = || drop(x);
```

2.
```
let mut x: T = ...;
let y: T = ...;
let cloture = || x = y;
```

3.
```
let x: &mut u32 = ...;
let cloture = || *x += 1;
```

4.
```
let x: &mut u32 = ...;
let mut temp = || *x += 1;
let cloture = &mut temp;
```

3 Closure Conversion

Lire le fichier `ocaml/src/ast.ml`.

Remplir le fichier `ocaml/src/closure_conv.ml`.

Nous allons implémenter l'opération de « closure conversion » pour un petit langage de programmation inspiré d'OCaml. Veuillez lire le fichier `ast.mli`, qui contient la définition du langage sur lequel nous travaillerons. À des fins de simplification, la traduction utilise le même langage en entrée et en sortie : ainsi, certaines parties de la définition de l'arbre de syntaxe ne sont pertinentes qu'avant la traduction, alors que d'autres ne sont pertinentes qu'après.

Le but de votre travail est de compléter le fichier `closure_conv.ml`, qui doit effectuer la closure conversion : cette traduction doit éliminer toutes les constructions `Fix`, en créant explicitement des clôtures avec la construction `Clos`, qui fait référence à une fonction globale, déclarée avec le constructeur `DFun`. Les environnements des clôtures sont stockés sous la forme de tableaux de valeurs, et peuvent être accédés avec la construction `EnvVar`, qui permet d'obtenir la valeur stockée à la position i de l'environnement de la clôture en cours d'exécution. L'argument de la clôture en cours d'exécution peut être accédé avec l'expression `Arg`, et on peut faire référence à la clôture en cours d'exécution avec la construction `Current_closure` (ceci est utile pour traiter les appels récursifs).

Ainsi, lorsque le programme contient une variable, il faut déterminer dans quel cas on se trouve :

- Si on est à top-level (pas dans une fonction), alors il ne faut rien faire ;
- S'il s'agit d'une variable locale introduite par la construction `let` dans la fonction courante, alors il ne faut rien faire non plus ;
- S'il s'agit d'une variable de la fonction ayant créé la clôture courante, alors il faut faire référence à l'environnement courant, à la bonne position ;
- S'il s'agit du paramètre de la fonction courante, alors il faut utiliser la construction `Arg` ;
- S'il s'agit d'un appel récursif à la fonction courante, alors il faut utiliser la construction `Current_closure` ;
- S'il s'agit d'une variable globale, alors on peut soit y faire référence directement par son nom, soit passer par l'environnement de la clôture courante, au choix.

Pour pouvoir facilement effectuer ce choix, on utilise le dictionnaire `env.vars`, qui fait correspondre à chaque nom de variable l'expression qui lui correspond après la traduction.

Une fois que vous aurez écrit la traduction, vous pourrez utiliser
`dune exec src/testing.exe`
afin de tester : l'exécutable interprète le programme donné en paramètre deux fois, avant et après la transformation, puis il compare les résultats.