Vers une traduction fonctionnelle linéaire des emprunts

Sidney Congard (Gallinette) - aux JFLA

1er février 2024

Échauffement

«Mutabilité xor partage»

Seul l'accès le plus récent à une valeur est accessible pour préserver cet invariant.

```
let mut a = 1;
let b = &mut a; // emprunte a
*b += 2;
// drop b
assert!(a == 3);
*b += 1; // ERREUR DE COMPILATION
```

Contexte

- Rust : notion de ressources + mutabilité restreinte
- Problème : quelle sémantique appropriée pour ce langage ?
- Contribution : traduction fonctionnelle linéaire des emprunts mutables et contraintes de région

Programme naïf

```
fn f<T>(i: bool, x: &mut T, y: &mut T, z: &mut T)
    -> (&mut T, &mut T)
{
    if i { (x, y) } else { (y, z) }
}
```

Programme naïf

```
fn f<T>(i: bool, x: &mut T, y: &mut T, z: &mut T)
    -> (&mut T, &mut T)
{
    if i { (x, y) } else { (y, z) }
}

let (mut u, mut v, mut w) = (10, 20, 30);
let (s, t) = f(true, &mut u, &mut v, &mut w);
*s += 1; *t += 2;
assert!(u == 11 && v == 22 && w == 30);
```

`f` ne compile pas : comment expliquer au *borrow checker* le lien entre `(s, t)` et `(u, v, w)` ?

Le borrow checker

Associe à chaque emprunt une «région» : origines possibles de sa valeur.

Accède à $a \Rightarrow$ Met fin aux régions $\{r \mid a \in r\} \Rightarrow$ Drop emprunts associés à chaque r.

Dépendances via inclusions : $r0 \subseteq r1 \Rightarrow a : r0$ vit plus longtemps que $b : r1$.

Vérification modulaire : requiert régions explicites dans les signatures de fonction.

Programme annoté

```
fn f<'a, T>(i: bool, x: &'a mut T, y: &'a mut T, z: &'a mut T)
    -> (&'a mut T, &'a mut T)
{
    if i { (x, y) } else { (y, z) }
}
let (mut u, mut v, mut w) = (10, 20, 30);
// r0 = {u} ∪ {v} ∪ {w}
let (s, t) = f(true, &mut u, &mut v, &mut w);
// s, t : r0
*s += 1; *t += 2;
// drop s, t
assert!(u == 11);
```

Problème de dépendance

```
fn f<'a, T>(i: bool, x: &'a mut T, y: &'a mut T, z: &'a mut T)
    -> (&'a mut T, &'a mut T)
{
    if i { (x, y) } else { (y, z) }
}
let (mut u, mut v, mut w) = (10, 20, 30);
let (s, t) = f(true, &mut u, &mut v, &mut w);
*s += 1;
// drop s, t
assert!(u == 11);
*t += 2; // ERREUR DE COMPILATION
assert!(v == 22 && w == 30);
```

Comment expliquer au borrow checker que t ne dépend pas de u ?

Programme final

```
fn f<'a, 'c, 'b: 'a+'c, T> /* b ⊆ a ∪ c */
    (i: bool, x: &'a mut T, y: &'b mut T, z: &'c mut T)
    -> (&'a mut T, &'c mut T)
{
    if i { (x, y) } else { (y, z) }
}
let (mut u, mut v, mut w) = (10, 20, 30);
let (s, t) = f(true, &mut u, &mut v, &mut w);
*s += 1;
// drop s
assert!(u == 11);
*t += 2;
// drop t
assert!(v == 22 && w == 30);
```

Vers une traduction fonctionnelle

Sémantique du programme obtenue via son interprétation dans un langage fonctionnel.

Approche Curry-Howard : obtient un programme fonctionnel de la preuve d'accessibilité.

"~~Est-ce que~~ **Comment** les valeurs sont accessibles ?"

Borrow checker	**Traduction**
Région $r = \{x_i, ..\}$ associée aux emprunts $y_j, ..$	«Route»= fonction linéaire $(y_j \otimes ..) \multimap (x_i \otimes ..)$
Fin de région r	Appel de route `let` $(y_i, ..) = r$
Emprunt	Valeur sous-jacente (état mutable local)
Drop d'emprunt $y : r$	Passage de y en argument à sa route r
Inclusion $a \subseteq b$	Dépendance fonctionnelle $b \to a$

Traduction de la première signature

Région $'a$ associée à x, y, z en entrée, à $ret.0, ret.1$ en sortie

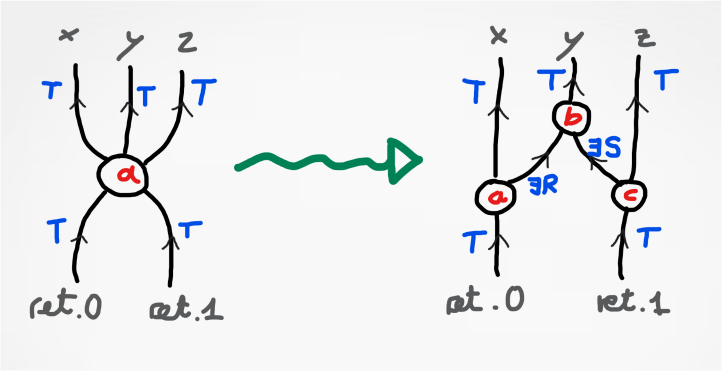
```
fn f<'a, T>(i: bool, x: &'a mut T, y: &'a mut T, z: &'a mut T)
    -> (&'a mut T, &'a mut T) { ... }
```

$\Rightarrow$ Route $(ret.0 \otimes ret.1) \multimap (x \otimes y \otimes z)$ en sortie

```
let f T (i: bool) (x y z: T):
    (T * T) * ((T * T)→(T * T * T)) = ...
```

Traduction de la seconde signature

```
fn f<'a, 'c, 'b: 'a+'c, T>(i: bool,
    x: &'a mut T, y: &'b mut T, z: &'c mut T) -> (&'a mut T, &'c mut T)
```




```
let f T (i: bool) (x y z: T):
    ((T * T) * ∃R.∃S. (T→(T * R)) * (T→(T * S)) * ((R * S)→T))
```

Traduction de l'appel de fonction

```
let (u, v, w) = (10, 20, 30) in
let ((s, t), a, c, b) = f(true, u, v, w) in
let s = s + 1 in
let (u, r0) = a s in (* drop s *)
assert(u == 11) in
let t = t + 2 in
let (w, r1) = c t in  (* drop t *)
let v = b (r0, r1) in
assert(v == 22 && w == 30)
```

Traduction du corps de la fonction

Traduit l'union des dépendances de chaque branche de manière fonctionnelle.

```
let f T (i: bool) (x y z: T):
    ((T * T) * ∃R.∃S. (T→(T * R)) * (T→(T * S)) * ((R * S)→T)) =
    if i { ((x, y), λx2. (x2, ()), λy2. (z, y2), λ((), y2). y2) }
    else { ((y, z), λy2. (x, y2), λz2. (z2, ()), λ(y2, ()). y2) }
```

Travaux futurs

- Préciser la correspondance et ses limites (coercions, équivalences de signatures, ...)
- Exploiter la linéarité pour la gestion des ressources (destructeurs, exceptions, effets, ...)
- Explorer d'autres fonctionnalités de Rust (emprunts imbriqués, polymorphisme, ... voir Aeneas [1] et papier associé [2])

1. ICFP 2022 - Rust Verification by Functional Translation, `github.com/AeneasVerif/aeneas`
2. JFLA 2024 - Towards a linear functional translation of borrowing

BONUS - Correspondance entre régions et routes

Région $'a$ associée à x, y, z en entrée, à $ret.0, ret.1$ en sortie

```
fn f<'a, T>(i: bool, x: &'a mut T, y: &'a mut T, z: &'a mut T)
    -> (&'a mut T, &'a mut T) { ... }
```

$\Rightarrow$ Route $(x \otimes y \otimes z) \multimap A$ en entrée, $(ret.0 \otimes ret.1) \multimap A$ en sortie

```
let f T A (i: bool) (x y z: T) (a: (T * T * T)→A):
    (T * T) * ((T * T)→A) = ...
```

$\simeq$ Route $(ret.0 \otimes ret.1) \multimap (x \otimes y \otimes z)$ en sortie

```
let f T (i: bool) (x y z: T):
    (T * T) * ((T * T)→(T * T * T)) = ...
```