Traduction fonctionnelle de programmes Rust

Contexte - Problème

Vérification de programmes bas-niveau et impératifs :

- Encodage lourd (allocations séparées, accessibles, …)
- Programmes annotés et preuves automatiques

Contexte - Objectif

Se baser sur les contraintes et garanties du système de type de Rust pour avoir une représentation fonctionnelle légère du programme.

Contexte - Approches similaires

- Electrolysis (2016) : le plus proche, assez limité
- Prusti (2019) : automatisation de règles de raisonnement sur la mémoire
- Creusot (2021) : encodage haut-niveau des références

Contexte - Rust

Langage bas-niveau : contrôle des performances et allocations, …

Langage fonctionnel : typeclasses, mutabilité explicite, ...

Son typage garantit l’absence de comportement indéfini.

Unsafe Rust : contourne localement le typage de Rust pour être plus expressif.

Aeneas ne vise que des programmes safe.

Contexte - Rust

Borrow checker : règles de typage dédiées au contrôle des références.

Soit plusieurs références partagées (&T), soit une référence mutable (&mut T).

Contexte - Rust

Borrow checker : règles de typage dédiées au contrôle des références.

Soit plusieurs références partagées (&T), soit une référence mutable (&mut T).

Nous allons nous focaliser sur les références mutables.

Contexte - Rust

Manipulation de références :

```rust
let mut x = 2;
let p = &mut x;
*p += 3;
// Fin de p.
assert!(x == 5);
```

Contexte - Rust

Régions associées aux références :

Explicitées par des paramètres de “lifetime”.

```
fn choose<'a, T>(e: bool, x: &'a mut T, y: &'a mut T)
    -> &'a mut T
{
    if e { x } else { y }
}
```

Contexte - Rust

Exemple d’appel de “choose” :

```
let mut x = 2;
let mut y = 6;
let p = choose(true, &mut x, &mut y);
*p += 3;
// drop p ~> récupère x et y.
assert!(x == 5);
```

Contexte - Rust

“re-borrowing” :

```
let mut x = 2;
let mut y = 6;
let p = choose(true, &mut x, &mut y);
inc(p); // inc(&mut *p)
inc(p);
inc(p);
assert!(x == 5);
```

```
fn inc(x: &mut u32) {
    *x += 1;
}
```

Contexte - Aeneas

LLBC = “Low-Level Borrow Calculus”

Exécution symbolique de LLBC ⇒ traduction de LLBC en λ-calcul

Principales limitations : safe Rust, pas de boucles, de références dans des énumérations ni de références imbriquées, pas de typeclasses.

Contexte - LLBC

LLBC = Rust simplifié et explicité :

- Copies, moves, drops, re-borrows, déréférencement explicites
- Ensemble de variables fixé par fonction
- Expressions limitées (programme séquentialisé) :

  g(f(x)) ⇝ { let y = f(x); g(y) }

Contexte - LLBC

```
let mut x = (1, 2);
// x ↦ (1, 2)

let y = &mut x;
// x ↦ loanᵐ l₀
// y ↦ borrowᵐ l₀ (1, 2)

let z = &mut (*y).0;
// x ↦ loanᵐ l₀
// y ↦ borrowᵐ l₀ (loanᵐ l₁, 2)
// z ↦ borrowᵐ l₁ 1

// Lire x ⇜ récupérer borrowᵐ l₀ ⇜ récupérer borrowᵐ l₁:
assert!(x == (1, 2));
// x ↦ (1, 2)
```

Par défaut, les variables sont inaccessibles :

$$v_i \mapsto \bot$$

Contexte - LLBC

```
let mut x = 2;
let mut y = 6;
let px = &mut x;
let py = &mut y;
// x ↦ loanᵐ l₀, y ↦ loanᵐ l₁, px ↦ borrowᵐ l₀ 2, py ↦ borrowᵐ l₁ 6

let p = choose(true, move px, move py);
// x ↦ loanᵐ l₀, y ↦ loanᵐ l₁, p ↦ borrowᵐ lᵣ (σ: u32)
// A { borrowᵐ l₀, borrowᵐ l₁, loanᵐ lᵣ }

*p += 3;
// ?
assert!(x == 5);
```

Contexte - LLBC

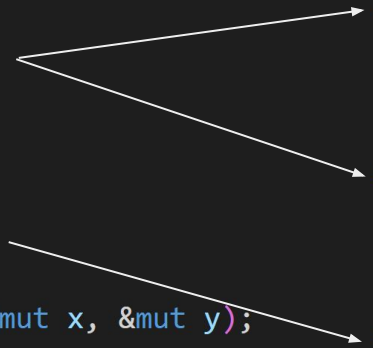
1. Valeur symbolique pour l'addition
2. Rend "p" à l'abstraction de région
3. Récupère x via l_0

```
// Étape 1:
// x ↦ loanᵐ l₀, y ↦ loanᵐ l₁, p ↦ borrowᵐ lᵣ (σ': u32)
// A { borrowᵐ l₀, borrowᵐ l₁, loanᵐ lᵣ }
// Étape 2:
// x ↦ loanᵐ l₀, y ↦ loanᵐ l₁, px' ↦ borrowᵐ l₀ σ₁, py' ↦ borrowᵐ l₁ σ₂
// Étape 3:
// x ↦ σ₁, y ↦ loanᵐ l₁, py' ↦ borrowᵐ l₁ σ₂
assert!(x == 5);
```

Contexte - Aeneas

```
fn choose<'a, T>(e: bool, x: &'a mut T, y: &'a mut T)
    -> &'a mut T
{
    if e { x } else { y }
}

fn call_choose() {
    let mut x = 2;
    let mut y = 6;
    let p = choose(true, &mut x, &mut y);
    *p += 3;
    assert!(x == 5);
}
```



```
let choose_fwd (t : Type) (b : bool) (x : t) (y : t)
    : result t =
  if b then Return x else Return y

let choose_back (t : Type) (b : bool) (x : t) (y : t) (ret : t)
    : result (t & t) =
  if b then Return (ret, y) else Return (x, ret)

let call_choose_fwd : result unit =
  p0 <-- choose_fwd i32 true 2 6; (* monadic let *)
  p1 <-- i32_add p0 3;
  (x0, y0) <-- choose_back i32 true 2 6 p1;
  massert (x0 = 5); (* monadic assert *)
  Return ()
```

Travaux - Constantes

Objectif : traduire des programmes avec des variables globales immutables.

```
const x: u32 = 1;
static y: u32 = x + 2;
```

Deux types de variables globales dans Rust :

- “static” : unique adresse garantie
- “const” : duplication / inlining possible

Travaux - Constantes

1. Parcours de l’AST : constantes, allocations taguées.
2. Réorganisation des déclarations : types, fonctions, constantes.
3. Récupération des déclarations externes (comme “u32::max”).
4. Actualisation du format LLBC : constantes, ADTs, assignements.
5. Traduction en fonctions sans arguments.
6. Évaluation des constantes hors de la monade d’erreur.

Travaux - Constantes

```
const x: u32 = 1;
static y: u32 = x + 2;
```



```
let x_body : result u32 = Return 1
let x_c : u32 = eval_global x_body

let y_body : result u32 =
    i <-- add_fwd x_c 2;
    Return i
let y_c : u32 = eval_global y_body
```

Travaux - Joins

Problème : duplication de la continuation des “match”.

```
if a { f() };
if b { g() };
h();
```

```
if a
then f ; if b
  then g ; h
  else h
else if b
  then g ; h
  else h
```

```
loop {
  if a { break };
  f();
}
g();
```

```
if a then g
else f ; if a
  then g
  else f ; if a
    then g
    else f ; ...
```

Travaux - Joins

Objectif : fusionner les environnements via une opération de “join”.

On doit obtenir un nouvel environnement en approximant ses références.

```
let mut x = 1;
let mut y = 2;
let p1;
let p2;
if e {
    p1 = &mut x;
    p2 = &mut y;
} else {
    p1 = &mut y;
    p2 = &mut x;
}
```

$$
\begin{aligned}
x &\mapsto \text{loan}^m\ l_0 \\
y &\mapsto \text{loan}^m\ l_1 \\
p1 &\mapsto \text{borrow}^m\ l_0\ 1 \\
p2 &\mapsto \text{borrow}^m\ l_1\ 2
\end{aligned}
\quad \cup \quad
\begin{aligned}
x &\mapsto \text{loan}^m\ l_2 \\
y &\mapsto \text{loan}^m\ l_3 \\
p1 &\mapsto \text{borrow}^m\ l_3\ 2 \\
p2 &\mapsto \text{borrow}^m\ l_2\ 1
\end{aligned}
$$

$$
\begin{aligned}
x &\mapsto (\text{loan}^m\ l_0) \cup (\text{loan}^m\ l_2) \\
y &\mapsto (\text{loan}^m\ l_1) \cup (\text{loan}^m\ l_3) \\
p1 &\mapsto (\text{borrow}^m\ l_0\ 1) \cup (\text{borrow}^m\ l_3\ 2) \\
p2 &\mapsto (\text{borrow}^m\ l_1\ 2) \cup (\text{borrow}^m\ l_2\ 1)
\end{aligned}
$$

Travaux - Joins

Union des dépendances. Règles de réécriture pour éliminer les “∪”.

$(a, b) \cup (c, d) \rightsquigarrow (a \cup c, b \cup d)$

$a \cup b, \text{loan} \notin a \cup b, \text{borrow} \notin a \cup b \rightsquigarrow s$

$a \cup b, \bot \in a, \bot \notin b \rightsquigarrow \text{erreur}$

$a \cup b, \text{loan} \in a \cup b, \text{borrow} \notin a \cup b \rightsquigarrow \text{loan } l, R \{ \text{borrow } l, \text{loans} \in a \cup b \ldots \}$

$(\text{borrow } l_0\ x) \cup (\text{borrow } l_1\ y) \rightsquigarrow \text{borrow } l_2\ s, R \{ \text{loan } l_2, \text{borrow } l_0, \text{borrow } l_1 \}$

Non-associatif : version n-aire.

Travaux - Joins

Fonctions backward pour de multiples loans :

```
x  ↦ loanᵐ l₄
y  ↦ (loanᵐ l₁) U (loanᵐ l₃)
p1 ↦ (borrowᵐ l₀ 1) U (borrowᵐ l₃ 2)
p2 ↦ (borrowᵐ l₁ 2) U (borrowᵐ l₂ 1)
R0 {
    loanᵐ l₀, loanᵐ l₂, borrowᵐ l₄,
    λl₀, l₂.match e with
    | true  → l₀
    | false → l₂
}
```

Cas général :

```
loan l₀, .. loan lₖ → borrow l,
λ(l₀, .. lₖ).match e with
| case 0 → v₀[lᵢ / loan lᵢ]
| ..
| case n → vₙ[lᵢ / loan lᵢ]
```

Travaux - Joins

Fonctions backward pour de multiples borrows :

```
x  ↦ loanᵐ l₄
y  ↦ (loanᵐ l₁) U (loanᵐ l₃)
p1 ↦ borrowᵐ l₅ s₀
p2 ↦ (borrowᵐ l₁ 2) U (borrowᵐ l₂ 1)
R0 { .. }
R1 {
    loanᵐ l₅, borrowᵐ l₀, borrowᵐ l₃,
    λl₅.match e with
    | true  → (l₅, 2)
    | false → (1, l₅)
}
```

Cas général :

```
loan l → borrow l₀ v₀, .. borrow lₙ vₙ,
λl.match e with
| case 0 → (l, v₁, .. vₙ)
| ..
| case k → (v₀, .. vₖ₋₁, l, vₖ₊₁, .. vₙ)
| ..
| case n → (v₀, .. vₙ₋₁, l)
```

Travaux - Joins

```
x  ↦ loanᵐ l₄
y  ↦ loanᵐ l₆
p1 ↦ borrowᵐ l₅ s₀
p2 ↦ borrowᵐ l₇ s₁
R0 (l₀, l₂) : l₄
  = match e with
    | true  → l₀
    | false → l₂
R1 l₅ : (l₀, l₃)
  = match e with
    | true  → (l₅, 2)
    | false → (1, l₅)
R2 (l₁, l₃) : l₆
  = match e with
    | true  → l₁
    | false → l₃
R3 l₇ : (l₁, l₂)
  = match e with
    | true  → (l₇, 1)
    | false → (2, l₇)
```

```
// Lire x ⇜ l₄ ⇜ R0:
// - l₀ ⇜ R1: l₅ ⇜ p1
// - l₂ ⇜ R3: l₇ ⇜ p2
// e == true:
x ⇜ l₄=x ⇜ l₀=x ⇜ l₅=x ⇜ p1=x
// e == false:
x ⇜ l₄=x ⇜ l₂=x ⇜ l₇=x ⇜ p2=x
```

Travaux - Joins

Simplification des régions :

- “inline” les dépendances dans les fonctions backward.
- normalise les termes avant de les exporter.

Traduction fonctionnelle :

Comment “join” des environnements avec leurs fonctions backward ?

Explorations - Interprétation des fonctions backward

```
x  ↦ loanᵐ l₄
y  ↦ loanᵐ l₆
p1 ↦ borrowᵐ l₅ 1
p2 ↦ borrowᵐ l₇ 2
R0 (l₀, l₂) : l₄ = l₀
R1 l₅ : (l₀, l₃) = (l₅, 2)
R2 (l₁, l₃) : l₆ = l₁
R3 l₇ : (l₁, l₂) = (l₇, 1)
```



```
let p1 = x in
let p2 = y in
let r0 = fun l0, l2 => l0 in
let r1 = fun l5 => (l5, y) in
let r2 = fun l1, l3 => l1 in
let r3 = fun l7 => (l7, x) in ...
```

```
x  ↦ loanᵐ l₄
y  ↦ loanᵐ l₆
p1 ↦ borrowᵐ l₅ 2
p2 ↦ borrowᵐ l₇ 1
R0 (l₀, l₂) : l₄ = l₂
R1 l₅ : (l₀, l₃) = (1, l₅)
R2 (l₁, l₃) : l₆ = l₃
R3 l₇ : (l₁, l₂) = (2, l₇)
```



```
let p1 = y in
let p2 = x in
let r0 = fun l0, l2 => l2 in
let r1 = fun l5 => (x, l5) in
let r2 = fun l1, l3 => l3 in
let r3 = fun l7 => (y, l7) in ...
```

Explorations - Typage des identifiants

Approche alternative pour les joins :

Typer les borrows/loans avec l’ensemble des identifiants possibles.

Vise à

- avoir des borrows dans des énumérations ou types récursifs.
- simplifier les valeurs à n’importe quel instant.
- contrôler la perte de précision lors de joins.

Le join s’applique aux types via l’union des ensembles d’identifiants.

Un paramètre de lifetime est un ensemble d’identifiants.

Explorations - Sémantique dénotationnelle

Problèmes : théories très syntaxiques et spécifiques.

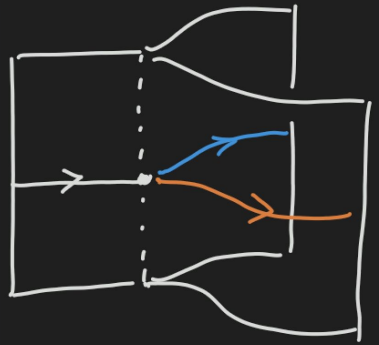
Objectif : chercher une structure qui éclaircit les invariants des opérations.

Prolonge l'interprétation d'une référence comme une valeur et sa continuation.

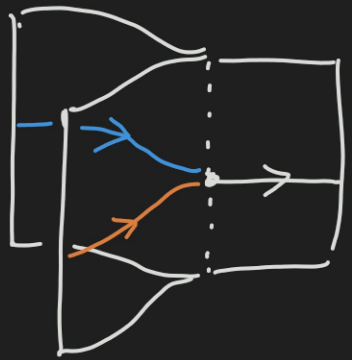
- Dénotation via opérations graphiques
- Invariants globaux : séquentialité du programme, validité des références
- Traduction = séquentialisation
- Règles locales pour ces invariants
- Regain des fonctionnalités de Rust
- Interprétation des types implémentés via du code unsafe

Explorations - Sémantique dénotationnelle

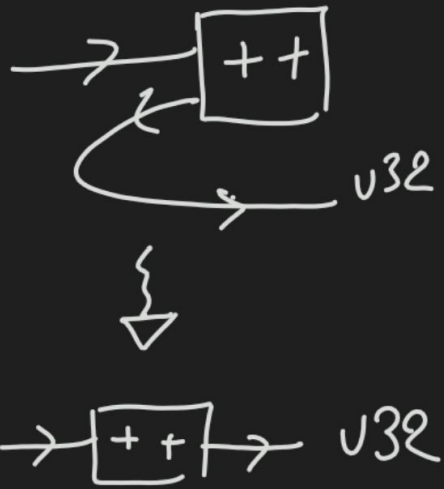
match :



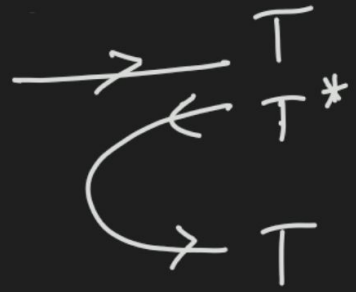
join :



traduction :

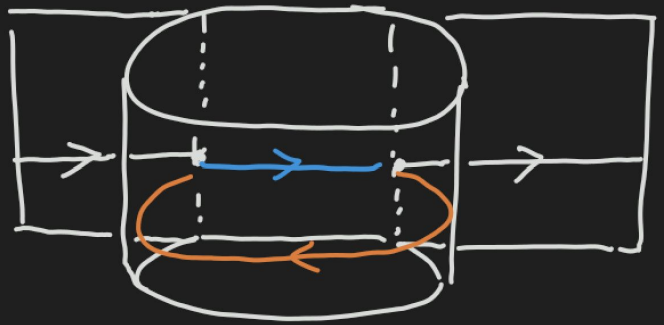


nouvelle référence :



drop de référence :

loop :



Merci !

Avez-vous des questions ?