Interprétation Abstraite du langage Rust

18.12.2025

Télécom SudParis, LIP6

Augustin PERRIN
augustin.perrin@telecom-sudparis.eu

Contents

1. Sous-ensemble du langage Rust (Mid-level Intermediate Representation)

1.1. Grammaire

On travaille sur la grammaire formelle suivante, du pseudo langage qu'on appelle λ_{MIR} :

Instruction	s	::=	`storage_live`$(v : \tau)$	*création de variable*
		|	`storage_dead`$(v : \tau)$	*suppression de variable*
		|	$e = e$	*affectation*
		|	$v =$ `&mut` v	*emprunt mutable*
		|	$v =$ `&`v	*référence partagée*
		|	$v =$ `move`(v)	*transfert d'appartenance*
		|	$s;s$	*chainage d'instructions*
		|	`loop`$(i \in \mathbb{N})\{s\}$	*boucle (imbrication $i \in \mathbb{N}$)*
		|	`break`$(i \in \mathbb{N})$	*terminateur de boucle*
		|	`if`$(c)\{s\}$ `else` $\{s\}$	*conditionnelle*
		|	`()`	*instruction vide*
Expression	e	::=	`copy`(e)	*copie*
		|	$v \in \mathcal{V}$	*variable*
		|	$z \in \mathbb{Z}$	*constante*
		|	$e \circledcirc e$	*opérateur binaire*
		|	$[z_1 \in \mathbb{Z} \cup \{-\infty\}; z_2 \in \mathbb{Z} \cup \{+\infty\}]$	*non déterminisme*
		|	$* e$	*déréférence*
Condition	c	::=	$e \circledast e$	*comparateur binaire*
		|	$!(c)$	*négation*
Type	τ	::=	`int`	*entier*
		|	`&mut` τ	*référence mutable*
		|	`&`τ	*référence partagée*
Opérateur	$\circledcirc$	::=	$+$	
		|	$-$	
		|	$*$	
		|	$/$	
		|	$\%$	
Comparateur	$\circledast$	::=	`<=`	
		|	`<`	
		|	`>=`	
		|	`>`	
		|	`==`	
		|	`!=`	

Ce sous ensemble modélise la représentation intermédiaire du compilateur `rustc` (MIR) réduite aux expressions et instructions de base pour l'analyse des références.

> **Remarque**
>
> L'expression $[z_1; z_2]$ représente formellement le non-déterminisme, ce qui correspond en Rust à un appel à `std::random::random()`.
>
> Les boucles sont non-conventionnelles et correspondent à celle de la MIR, où $i \in \mathbb{N}$ représente un identifieur unique correspondant informellement au niveau d'imbrication. Par exemple :

```
loop(0) { // (L0)
    loop(1) {
        if (i % 2 == 0) {
            break 0 // (L1)
        } else {
            ()
        }
    }
}
```

ici, à la ligne `(L1)` on a `break(0)` qui correspond à la terminaison de la boucle `(L0)`.

Exemple

exemple de programme :

```
storage_live(local6 : int);
storage_live(a : int);
storage_live(b : int);
storage_live(c : &mut int);
a = 0;
b = 0;
if ([-∞;+∞] != 0) {
    c = &mut a
} else {
    c = &mut b
};
local6 = copy(*c) + 1;
*c = move(local6);
storage_dead(local6 : int);
storage_dead(c);
storage_dead(a);
storage_dead(b)
```

correspondant au programme suivant en Rust :

```
let mut a = 0;
let mut b = 0;
let c = if std::random::random() {
    &mut a
} else {
    &mut b
};
*c += 1;
```

1.2. Typage

On présente ici le système de type de λ_{MIR}, par induction structurelle sur les expressions. On se munit d'un jugement de type $\Gamma : \mathcal{V} \to \mathrm{T}$ (en notant T l'ensemble des types).

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}(T_0) \qquad \frac{}{\Gamma \vdash z \in \mathbb{Z} : \texttt{int}}(T_1)$$

$$\frac{}{\Gamma \vdash [z_1 \in \mathbb{Z} \cup \{-\infty\}; z_2 \in \mathbb{Z} \cup \{+\infty\}] : \texttt{int}}(T_2) \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \circledcirc e_2 : \texttt{int}}(T_3)$$

$$\frac{\Gamma \vdash e : \texttt{\&mut } \tau}{\Gamma \vdash (* e) : \tau}(T_4) \qquad \frac{\Gamma \vdash e : \texttt{\&}\tau}{\Gamma \vdash (* e) : \tau}(T_5) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{copy}(e) : \tau}(T_6)$$

et pour les instructions, on ajoute les règles suivantes, en notant $\Gamma, s \vdash \texttt{ok}$ pour dire que s est bien typé sous le contexte Γ :

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad \Gamma \vdash s_1 : \texttt{ok} \quad \Gamma \vdash s_2 : \texttt{ok}}{\Gamma \vdash \texttt{if } (e_1 \circledast e_2)\{s_1\} \texttt{ else } \{s_2\} : \texttt{ok}}(S_0)$$

$$\frac{\Gamma, e_1 : \texttt{int} \vdash e_2 : \texttt{int}}{\Gamma, e_1 : \texttt{int} \vdash (e_1 = e_2) : \texttt{ok}}(S_1) \qquad \frac{\Gamma, v : \tau \vdash x : \tau}{\Gamma, v : \tau \vdash (x = \texttt{move}(v)) : \texttt{ok}}(S_2)$$

$$\frac{\Gamma, x : \tau \vdash u : \texttt{\&mut } \tau}{\Gamma, x : \tau \vdash (u = \texttt{\&mut } x) : \texttt{ok}}(S_3) \qquad \frac{\Gamma, x : \tau \vdash u : \texttt{\&}\tau}{\Gamma, x : \tau \vdash (u = \texttt{\&}x) : \texttt{ok}}(S_4)$$

$$\frac{\Gamma \vdash s : \texttt{ok}}{\Gamma \vdash \texttt{loop}(i \in \mathbb{N})\{s\} : \texttt{ok}}(S_5)$$

$$\frac{\Gamma, x : \tau \vdash s : \texttt{ok}}{\Gamma \vdash \texttt{storage\_live}(x : \tau); s : \texttt{ok}}(S_6) \qquad \frac{\Gamma \vdash s : \texttt{ok}}{\Gamma, x : \tau \vdash \texttt{storage\_dead}(x : \tau); s : \texttt{ok}}(S_7)$$

$$\frac{}{\Gamma \vdash \texttt{break}(i \in \mathbb{N}) : \texttt{ok}}(S_8) \qquad \frac{}{\Gamma \vdash \texttt{()} : \texttt{ok}}(S_9) \qquad \frac{\Gamma \vdash s_1 : \texttt{ok} \quad \Gamma \vdash s_2 : \texttt{ok}}{\Gamma \vdash s_1; s_2 : \texttt{ok}}(S_{10})$$

Remarque

Ce système de type ne vérifie pas les emprunts. Ces derniers sont vérifiés dans une passe de compilation séparée après le typage appelée *borrow checking*.

2. Sémantique transitionnelle des programmes

Définition 1

Un état $\mathcal{S}$ d'un programme Rust consiste en une fonction des variables du programme $\mathcal{V}$ vers les entiers mathématiques (scalaire) et les variables (références), i.e.:

$$\mathcal{S} \triangleq \mathcal{V} \to \mathbb{Z} \cup \mathcal{P}^{\text{tr}}$$

En notant $\mathcal{P}^{\text{tr}} \triangleq \mathcal{V} \cup \{\text{INVALID}, \text{UNINIT}\}$

Cette définition ne modélise volontairement pas d'allocations dynamiques sur le tas, ce qui simplifie le domaine sémantique à l'étude. Ces allocations seront ajoutées plus tard.

Définition 2

Notre sémantique concrète peut être séparée entre la sémantique concrète numérique (les variables de type `int`) et la sémantique concrète des références (les variables de type `&`τ, et `&mut` τ). On note le domaine concret numérique :

$$(\mathcal{D}, \subseteq, \cup, \cap, \emptyset, \mathcal{P}(\mathcal{V} \to \mathbb{Z})) \text{ où } \mathcal{D} \triangleq \bigcup_{S \in \mathcal{P}(\mathcal{S})} \left\{ \rho_{|S_{\texttt{int}}} \mid \rho \in S \right\}$$

En notant pour tout $S \in \mathcal{P}(\mathcal{S})$:

$$S_{\texttt{int}} \triangleq \{x \mid x \in S, x : \texttt{int}\}$$

Exemple

Dans l'exemple précédent, on aurait au début du programme :

$$(a \to \text{UNINIT}, b \to \text{UNINIT}, c \to \text{UNINIT})$$

Puis à la fin du programme avant les désallocations l'environnement suivant :

$$(a \to 0, b \to 1, c \to b)$$

si `std::random::random()` a été évalué à un scalaire supérieur à 0, et :

$$(a \to 1, b \to 0, c \to a)$$

sinon. Puis après les désallocations :

$$(a \to \text{INVALID}, b \to \text{INVALID}, c \to \text{INVALID})$$

2.1. Sémantique des expressions

La sémantique d'une expression e est dénotée $\mathbb{E}[\![e]\!] : \mathcal{S} \to \mathcal{P}(\mathbb{Z}) \cup \mathcal{P}(\mathcal{P}^{tr})$. Pour un état d'entrée donné, l'expression est évaluée dans l'ensemble des valeurs de sorties possibles (à cause du non déterminisme). Une variable est évaluée à l'aide de l'état d'entrée σ. Si la variable n'est pas définie, on renvoie l'ensemble vide et on considère cet état comme une erreur. On disjoint l'union entre les scalaires et les variables car on part du principe que le programme est bien typé, donc une expression a un type unique et ne peut pas s'évaluer en un ensemble de valeurs contenant à la fois des entiers et des références. On a alors :

$$\mathbb{E}[\![v \in \mathcal{V}]\!]\sigma \triangleq \{\sigma(v)\}$$

$$\mathbb{E}[\![z \in \mathbb{Z}]\!]\sigma \triangleq \{z\}$$

$$\mathbb{E}[\![[z_1; z_2]]\!]\sigma \triangleq \{z \in \mathbb{Z} \mid z_1 \leq z \leq z_2\}$$

$$\mathbb{E}[\![e_1 \circledcirc e_2]\!]\sigma \triangleq \mathbb{E}[\![e_1]\!]\sigma \circledcirc \mathbb{E}[\![e_2]\!]\sigma$$

Avec :

$$X \circledcirc Y \triangleq \{x \circledcirc y \mid x \in X, y \in Y\} \forall \circledcirc \in \{+, -, *\}$$

$$X / Y \triangleq \{x / y \mid x \in X, y \in Y, y \neq 0\}$$

$$X \% Y \triangleq \{x \% y \mid x \in X, y \in Y, y \neq 0\}$$

Puis pour les expressions liées aux références :

$$\mathbb{E}[\![* v]\!]\sigma \triangleq \bigcup_{\substack{u \in \sigma(v), \\ u \neq \text{INVALID}, \\ u \neq \text{UNINIT}}} \{\sigma(u)\}$$

$$\mathbb{E}[\![\texttt{copy}(e)]\!]\sigma \triangleq \mathbb{E}[\![e]\!]\sigma$$

> **Remarque**
>
> le `move` est traité comme une instruction car il n'a une sémantique attribuée que pour l'affectation d'un `move`, il en va de même pour les emprunts mutables et partagés, les expressions ci-dessous sont donc invalides :
>
> $$x = \texttt{move}(v) + 1$$
> $$x = \texttt{\&mut } v + 1$$

2.2. Sémantique des instructions

A cause du non-déterminisme des expressions, on peut se retrouver avec une affectation qui transforme un état du programme en plusieurs. On note $\mathbb{S}[\![s]\!] : \mathcal{S} \to \mathcal{P}(\mathcal{S})$ la sémantique d'une instruction s. Pour faciliter leur composition, on définit une extension $\mathbb{S}[\![s]\!] : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$. La sémantique d'une instruction, étant donné l'ensemble des états possibles avant l'instruction, associe l'ensemble des états possibles après l'instruction. Pour Σ un ensemble d'états, on a alors :

$$\mathbb{S}[\![\texttt{storage\_live}(v : \texttt{int})]\!]\Sigma \triangleq \{\sigma[v \to z] \mid \sigma \in \Sigma, z \in \mathbb{Z}\}$$

$$\mathbb{S}[\![x = e]\!]\Sigma \triangleq \{\sigma[x \to v] \mid \sigma \in \Sigma, v \in \mathbb{E}[\![e]\!]\sigma\}$$

$$\mathbb{S}[\![x = \texttt{move}(v)]\!]\Sigma \triangleq \{\sigma[x \to \sigma(v), v \to \text{UNINIT}] \mid \sigma \in \Sigma\}$$

$$\mathbb{S}[\![s_1; s_2]\!] \triangleq \mathbb{S}[\![s_2]\!] \circ \mathbb{S}[\![s_1]\!]$$

$$\mathbb{S}[\![\texttt{storage\_dead}(v : \texttt{int})]\!]\Sigma \triangleq \{\sigma[v \to \text{INVALID}, \forall x \in \mathcal{V}, \text{tq. } \sigma(x) = v, x \to \text{INVALID}] \mid \sigma \in \Sigma\}$$

Pour les références, on ajoute :

$$\mathbb{S}[\![\texttt{storage\_live}(v : \texttt{\&}\tau)]\!]\Sigma \triangleq \{\sigma[v \to \text{UNINIT}] \mid \sigma \in \Sigma\}$$

$$\mathbb{S}[\![* x = e]\!]\Sigma \triangleq \{\sigma[u \to v] \mid \sigma \in \Sigma, v \in \mathbb{E}[\![e]\!]\sigma, u \in \mathbb{E}[\![x]\!]\sigma, u \neq \text{INVALID}, u \neq \text{UNINIT}\}$$

$$\mathbb{S}[\![\texttt{storage\_dead}(v : \texttt{\&}\tau)]\!]\Sigma \triangleq \{\sigma[v \to \text{INVALID}, \forall x \in \mathcal{V}, \text{tq. } \sigma(x) = v, x \to \text{INVALID}] \mid \sigma \in \Sigma\}$$

$$\mathbb{S}[\![x = \texttt{\&}v]\!]\Sigma \triangleq \{\sigma[x \to v] \mid \sigma \in \Sigma\}$$

$$\mathbb{S}[\![x = \texttt{\&mut } v]\!]\Sigma \triangleq \{\sigma[x \to v] \mid \sigma \in \Sigma\}$$

2.2.1. Sémantique des instructions conditionnelles

Pour les instructions conditionnelles, on définit un opérateur de filtrage conditionnel $\mathbb{C}[\![c]\!]$:

$$\mathbb{C}[\![e_1 \circledast e_2]\!] \triangleq \{\sigma \in \Sigma \mid \exists v_1 \in \mathbb{E}[\![e_1]\!]\sigma, \exists v_2 \in \mathbb{E}[\![e_2]\!]\sigma, v_1 \circledast v_2\}$$

On peut ensuite définir :

$$\mathbb{S}[\![\texttt{if}(c)\{s_t\}\texttt{else}\{s_f\}]\!] \triangleq \mathbb{S}[\![s_t]\!] \circ \mathbb{C}[\![c]\!] \cup \mathbb{S}[\![s_f]\!] \circ \mathbb{C}[\![\neg c]\!]$$

2.2.2. Sémantique des boucles

Pour les boucles, on itère sur le corps de la boucle à l'infini puis on garde les états qui ont quitté la boucle (i.e., qui ont croisé un `break` au label $i \in \mathbb{N}$ correspondant à la boucle), on doit donc d'abord définir un état concret $\langle \Sigma, \text{out} \rangle \in \mathcal{P}(\mathcal{S}) \times (\mathbb{N} \to \mathcal{P}(\mathcal{S}))$, où pour $i \in \mathbb{N}$, $\text{out}(i)$ correspond aux états collectés à la fin des $\texttt{break}(i)$, tel que :

Pour tout $s \in \texttt{Instruction}, s \neq \texttt{break}, s \neq \texttt{loop}$:

$$\mathbb{S}[\![s]\!]\langle \Sigma, \text{out} \rangle \triangleq \langle \mathbb{S}[\![s]\!]\Sigma, \text{out} \rangle$$

Puis $\texttt{break}(i \in \mathbb{N})$ déplace tous les états courants dans $\text{out}(i)$:

$$\mathbb{S}[\![\texttt{break}(i \in \mathbb{N})]\!]\langle \Sigma, \text{out} \rangle \triangleq \langle \emptyset, \text{out}[i \to \text{out}(i) \cup \Sigma] \rangle$$

Et à la fin d'une boucle $\texttt{loop}(i \in \mathbb{N})$, les états sortants sont les $\text{out}(i)$:

$$\mathbb{S}[\![\texttt{loop}(i \in \mathbb{N})\{s\}]\!]\langle \Sigma, \text{out} \rangle \triangleq \texttt{let } \langle \Sigma', \text{out}' \rangle = \text{lfp}(\mathbb{S}[\![s]\!])\langle \Sigma, \text{out} \rangle \texttt{ in } \langle \text{out}'(i), \text{out}'[i \to \emptyset] \rangle$$

$$\text{avec : } \text{lfp}(\mathbb{S}[\![s]\!])\langle \Sigma, \text{out} \rangle \triangleq \lim_{n \to \infty} F_n \langle \Sigma, \text{out} \rangle,$$

$$\text{où } F_n \langle \Sigma, \text{out} \rangle \triangleq \begin{cases} \langle \Sigma, \text{out} \rangle \text{ si } n = 0 \\ \mathbb{S}[\![s]\!](F_{n-1}\langle \Sigma, \text{out} \rangle) \cup F_{n-1}\langle \Sigma, \text{out} \rangle \text{ sinon} \end{cases}$$

3. Sémantique abstraite

On prend un domaine abstrait numérique $\mathcal{D}^\sharp$ muni de :

1. un ensemble $\mathcal{D}^\sharp$ de valeurs abstraites,
2. un ordre partiel muni d'un algorithme effectif $\sqsubseteq^\sharp$ sur $\mathcal{D}^\sharp$,
3. une fonction de concrétisation $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$,
4. un plus petit élément $\bot^\sharp \in \mathcal{D}^\sharp$ et un plus grand élément $\top^\sharp \in \mathcal{D}^\sharp$,
5. des abstractions effectives et **sûres** des conditions et des affectations arithmétiques :

$$\mathbb{S}^\sharp[\![v = e]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$$

$$\mathbb{C}^\sharp[\![e_1 \circledast e_2]\!] : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$$

telles que :

$$\forall \sigma^\sharp \in \mathcal{D}^\sharp : (\mathbb{S}[\![v = e]\!] \circ \gamma)\sigma^\sharp \subseteq \left(\gamma \circ \mathbb{S}^\sharp[\![v = e]\!]\right)\sigma^\sharp$$

$$\forall \sigma^\sharp \in \mathcal{D}^\sharp : (\mathbb{C}[\![e_1 \circledast e_2]\!] \circ \gamma)\sigma^\sharp \subseteq \left(\gamma \circ \mathbb{C}^\sharp[\![e_1 \circledast e_2]\!]\right)\sigma^\sharp$$

6. des abstractions effectives et **sûres** de l'union ensembliste et de l'intersection :

$$\cup^\sharp : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$$

$$\cap^\sharp : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$$

telles que :

$$\forall \sigma_1^\sharp, \sigma_2^\sharp \in \mathcal{D}^\sharp : \gamma\left(\sigma_1^\sharp\right) \cup \gamma\left(\sigma_2^\sharp\right) \subseteq \gamma\left(\sigma_1^\sharp \cup^\sharp \sigma_2^\sharp\right)$$

$$\forall \sigma_1^\sharp, \sigma_2^\sharp \in \mathcal{D}^\sharp : \gamma\left(\sigma_1^\sharp\right) \cap \gamma\left(\sigma_2^\sharp\right) \subseteq \gamma\left(\sigma_1^\sharp \cap^\sharp \sigma_2^\sharp\right)$$

7. un opérateur d'élargissement ∇.

A partir de ce domaine abstrait, on définit la sémantique abstraite des programmes ci-dessous.

3.1. Sémantique abstraite des instructions

On peut définir la sémantique abstraite des instructions par induction avec quelques changements notables, pour $\sigma^\sharp \in \mathcal{D}^\sharp$, $\text{out}^\sharp \in \mathbb{N} \to \mathcal{D}^\sharp$, pour tout $s \in \texttt{Instruction}$, $s \neq \texttt{break}$, $s \neq \texttt{loop}$:

$$\mathbb{S}^\sharp[\![s]\!]\langle\sigma^\sharp, \text{out}^\sharp\rangle \triangleq \langle \mathbb{S}^\sharp[\![s]\!]\sigma^\sharp, \text{out}^\sharp\rangle$$

Puis :

$$\mathbb{S}^\sharp[\![\texttt{break}(i \in \mathbb{N})]\!]\langle\sigma^\sharp, \text{out}^\sharp\rangle \triangleq \langle \bot^\sharp, \text{out}^\sharp\left[i \to \text{out}^\sharp(i) \cup^\sharp \sigma^\sharp\right]\rangle$$

$$\mathbb{S}^\sharp[\![\texttt{loop}(i \in \mathbb{N})\{s\}]\!]\langle\sigma^\sharp, \text{out}^\sharp\rangle \triangleq \texttt{let } \langle\sigma^{\sharp\prime}, \text{out}^{\sharp\prime}\rangle = \text{lfp}\left(\mathbb{S}^\sharp[\![s]\!]\right)\langle\sigma^\sharp, \text{out}^\sharp\rangle \texttt{ in } \langle \text{out}^{\sharp\prime}(i), \text{out}^{\sharp\prime}\left[i \to \bot^\sharp\right]\rangle$$

$$\text{avec : } \text{lfp}\left(\mathbb{S}^\sharp[\![s]\!]\right)\langle\sigma^\sharp, \text{out}^\sharp\rangle \triangleq \lim_{n\to\infty} F_n^\sharp\langle\sigma^\sharp, \text{out}^\sharp\rangle$$

$$\text{où } F_n^\sharp\langle X^\sharp, \text{out}^\sharp\rangle \triangleq \begin{cases} \langle X^\sharp, \text{out}^\sharp\rangle \text{ si } n = 0 \\ F_{n-1}^\sharp\langle X^\sharp, \text{out}^\sharp\rangle \nabla \mathbb{S}^\sharp[\![s]\!]\left(F_{n-1}^\sharp\langle X^\sharp, \text{out}^\sharp\rangle\right) \text{ sinon} \end{cases}$$

$$\mathbb{S}^\sharp[\![\texttt{if}(c)\{s_t\}\texttt{else}\{s_f\}]\!] \triangleq \mathbb{S}^\sharp[\![s_t]\!] \circ \mathbb{C}^\sharp[\![c]\!] \cup^\sharp \mathbb{S}^\sharp[\![s_f]\!] \circ \mathbb{C}^\sharp[\![\neg c]\!]$$

Théorème

$\mathbb{S}^\sharp[\![p]\!]$ termine toujours et est **sûre** :

$$\forall p \in \mathfrak{S}, \sigma^\sharp \in \mathcal{D}^\sharp : \mathbb{S}[\![p]\!]\left(\gamma\left(\sigma^\sharp\right)\right) \subseteq \gamma\left(\mathbb{S}^\sharp[\![p]\!]\sigma^\sharp\right)$$

En notant $\mathfrak{S}$ l'ensemble des programmes (induits par la grammaire Section 1.1).

La définition de l'opérateur d'élargissement ∇ assure que la limite $F_n^\sharp$ est toujours atteinte après un nombre fini d'itérations. Le résultat de l'analyse est **sûr** : c'est la composition d'abstractions sûres.

4. Domaine abstrait des références

On aimerait désormais raisonner de manière **automatique** sur les références. Ces dernières sont intuitivement moins compliquées que les pointeurs du langage C du fait que les emplacements en mémoire sont indivisibles. On a aussi l'information du *borrow checker* modélisée en Section 4.2.2 qui nous indique qu'une zone mémoire doit avoir au plus un seul emprunt mutable à la fois, qui est temporairement "responsable" de la zone. Cette information induit une adaptation des sémantiques abstraite et concrète.

4.1. Exemple motivateur

On s'intéresse au programme suivant :

Programme λ_{MIR}	Programme Rust
`storage_live(a : int);`	`let a = std::random::random() % 10;`
`storage_live(b : int);`	`let b = std::random::random() % 10;`
`a = [0, 10];`	`let ma = &mut a;`
`b = [0, 10];`	`let mb = &mut b;`
`storage_live(ma : &mut int);`	`let mc = if *ma >= *mb {`

Programme λ_{MIR}	Programme Rust
`storage_live(mb : &mut int);`	`    ma`
`ma = &mut a;`	`} else {`
`mb = &mut b;`	`    mb`
`storage_live(mc : &mut int);`	`};`
`if (*ma >= *mb) {`	`*mc += 1;`
`    mc = ma;`	`[...]`
`} else {`	
`    mc = mb;`	
`};`	
`*mc += 1;`	
`[...]`	

Remarque

On choisit un emprunt selon les valeurs de a et b qui sont non déterministes, et on incrémente le déréférencement de cet emprunt.

On aimerait être capable de prouver que le programme vérifie la post-condition $a \neq b$.

La difficulté réside ici dans le choix de la zone mémoire au moment de l'exécution de :

`*mc += 1;`

Ce choix peut être entièrement non déterministe si l'on change le test `*ma >= *mb`, auquel cas pour rester sûr en utilisant le domaine des intervalles, on devra modifier l'état mémoire de manière à avoir :

$$\sigma = (a \to [0; 11], b \to [0; 11])$$

Dans ce cas, l'invariant $a \neq b$ n'est plus prouvé. Et dans le cas où la condition d'emprunt est déterministe (comme sur cet exemple), on souhaiterait garder l'information que :

$$\texttt{mc} = \begin{cases} \texttt{ma si *ma >= *mb} \\ \texttt{mb sinon} \end{cases}$$

De cette manière, on tirerait une meilleure précision que sur un domaine classique de pointeurs du fait que les références Rust sont généralement plus strictes dans leur définition : en effet, un pointeur en C peut pointer vers un nombre potentiellement très dense de variables, tandis qu'une référence Rust ne pointera jamais sur plus qu'un nombre statique prédéfini à la compilation de variables.

Cela est dû à la sémantique de possession, vérifiée par le *borrow checker*, où par exemple :

```
let mut a = [0;4096]; // définition d'un tableau de taille 4096
let b = &mut a[0];
let c = &mut a[1];
[...] // opérations sur b
```

ne compile pas, en affichant l'erreur suivante :

```
error[E0499]: cannot borrow `a[_]` as mutable more than once at a time
```

Ces emprunts sont un problème uniquement du fait qu'ils soient mutables. Dans le cas d'une référence partagée, on peut réaliser autant d'emprunts que l'on veut même si la philosophie générale de la programmation en Rust encourage un faible nombre de références partagées. Cette information nous permet de créer un modèle plus précis pour modéliser les références, mais aussi vérifier l'absence de bugs du compilateur en s'intéressant à sa sémantique d'emprunts formalisée Section 4.2.2.

4.2. Domaine de base $\mathcal{E}^\sharp$

4.2.1. Etat mémoire abstrait $\mathcal{E}$

Définition 3

On définit le treillis $(\mathcal{P}(\mathcal{E}), \preccurlyeq, \cup)$ avec :

$$\mathcal{E} \triangleq \bigcup_{C \subseteq \mathcal{V}} \{\langle C, \rho\rangle \mid \rho \in C \to \mathbb{Z} \cup \mathcal{P}^{\text{tr}}\}$$

Avec l'ordre partiel suivant, pour $X, X' \in \mathcal{P}(\mathcal{E})$:

$$X \preccurlyeq X' \overset{\Delta}{\iff} \forall \langle C, \rho\rangle \in X, \exists \langle C', \rho'\rangle \in X' : (C' \subseteq C) \wedge \left(\rho' = \rho_{|C'}\right)$$

où $\rho_{|C'}$ dénote la restriction de ρ à C', et ρ représente un environnement scalaire.
Cet ensemble définit un état mémoire abstrait.

Cette sémantique abstraite est inspirée par l'analyse de portabilité [1] de l'analyseur Mopsa.

Exemple

Dans ce programme :

```
storage_live(a : int);
storage_live(b : int);
storage_live(c : &mut int);
a = 0;
b = 1;
if ([0;10] == 0) {
    c = &mut a
} else {
    c = &mut b
}
```

On a :

$$\mathcal{R} = \{\langle \{a, b, c\}, \rho_1\rangle, \langle \{a, b, c\}, \rho_2\rangle\}$$

Avec $\rho_1[c \to a, a \to 0, b \to 1], \rho_2[c \to b, a \to 0, b \to 1]$.

Une propriété abstraite $X \in \mathcal{P}(\mathcal{E})$ représente l'ensemble concret d'états $\gamma_{\mathcal{V}}(X) \in \mathcal{P}(\mathcal{V} \to \mathbb{Z} \cup \mathcal{P}^{\text{tr}})$:

$$\gamma_{\mathcal{V}}(X) = \left\{\mu : \mathcal{V} \to \mathbb{Z} \cup \mathcal{P}^{\text{tr}} \mid \exists \langle C, \rho\rangle \in X : \mu_{|C} = \rho\right\}$$

La suppression de référence est toujours **sûre** : elle dénote une perte d'information. Il est aussi possible d'ajouter des nouvelles références simplement : l'ajout d'une nouvelle référence initialisée à $\top$ n'a pas d'effet de bord sur le reste de l'environnement et reste donc sûr.

4.2.2. Sémantique abstraite du *Borrow Checker*

Le *borrow checker* a vérifié à la compilation jusqu'à la MIR que chaque espace mémoire a un unique propriétaire, qui est le seul à pouvoir éventuellement le modifier. Une analyse dynamique des emprunts a été formalisée par le modèle de *Stacked Borrows* [2] dont la sémantique qui suit est inspirée. Formellement, on ajoute $\psi : \mathcal{V} \to \text{List}(I_\mathcal{V})$ à l'état abstrait, avec :

$$I_\mathcal{V} \triangleq \{\text{unique}(v), v \in \mathcal{V}\} \cup \{\text{shared}(X), X \subseteq \mathcal{V}\}$$

$$\text{List}(I) \triangleq \text{Nil} \mid \text{Cons}(t,q), t \in I, q \in \text{List}(I)$$

Où, pour $v \in \mathcal{V}$, $\psi(x) = \text{Cons}(\text{unique}(v), q)$ représente l'appartenance de l'espace mémoire correspondant à x par v, et $\psi(x) = \text{Cons}(\text{shared}(\{v\}), q)$ représente informellement un emprunt en lecture seule. On appelle $\psi(x)$ la pile d'emprunts correspondant à l'espace mémoire x. Par exemple :

```
let mut a = 0;  // (L1)
let b = &mut a; // (L2)
let c = &mut b; // (L3)
let d = &c;     // (L4)
```

après (L4), on aura :

$$\psi(a) = \text{Cons}(\text{shared}(d), \text{Cons}(\text{unique}(c), \text{Cons}(\text{unique}(b), \text{Cons}(\text{unique}(a), \text{Nil}))))$$

que l'on note dans la suite par abus $\psi(a) = [\text{shared}(\{d\}), \text{unique}(c), \text{unique}(b), \text{unique}(a)]$.
On définit aussi par induction, pour $x, v \in \mathcal{V}$, les opérateurs :

$$x \in \psi(v) \overset{\Delta}{\iff} (\psi(v) = \text{Cons}(x, q)) \vee (\text{let Cons}(t, q) = \psi(v) \text{ in } x \in q)$$

$$\text{until}_{\text{mut}}(x, \psi(v)) \triangleq \begin{cases} \text{until}_{\text{mut}}(x, q) \text{ si } \psi(v) = \text{Cons}(t, q), t \neq \text{unique}(x) \\ \psi(v) \text{ si } \psi(v) = \text{Cons}(\text{unique}(x), q) \\ \text{Nil si } \psi(v) = \text{Nil} \end{cases}$$

$$\text{until}_{\text{shr}}(x, \psi(v)) \triangleq \begin{cases} \psi(v) \text{ si } \psi(v) = \text{Cons}(\text{shared}(X), q), x \in X \\ \text{Nil si } \psi(v) = \text{Nil} \\ \text{until}_{\text{shr}}(x, q) \text{ avec } \psi(v) = \text{Cons}(t, q) \text{ sinon} \end{cases}$$

Les opérateurs until vont informellement supprimer tous les emprunts se situant avant x dans la pile d'emprunts $\psi(v)$, si x appartient à cette pile. Avec la sémantique transitionnelle suivante, pour l'état concret $\langle C, \psi, \rho \rangle$:

$$\mathbb{S}[\![\texttt{storage\_live}(x)]\!]\langle C, \psi, \rho \rangle \triangleq \{\langle C, \psi[x \to [\text{unique}(x)]], \rho \rangle\}$$

Par défaut, $x \in \mathcal{V}$ est le seul alias à avoir des droits de lecture et d'écriture sur son espace mémoire. Puis les opérations d'emprunt mutable et partagé placent sur la pile d'emprunts de l'espace mémoire sous-jacent la nouvelle référence :

$$\mathbb{S}[\![t = \texttt{\&mut } x]\!]\langle C, \psi, \rho\rangle \triangleq \{\langle C, \psi[x \to \text{Cons}(\text{unique}(t), \text{until}_{\text{mut}}(x, \psi(x)))], \rho\rangle\}$$

$$\mathbb{S}[\![x = \texttt{\&}v]\!]\langle C, \psi, \rho\rangle \triangleq \left\{ \langle C, \psi \left[x \to \begin{cases} \text{Cons}(\text{shared}(\{x\} \cup X), q) \\ \text{si } \text{until}_{\text{shr}}(v, \psi(v)) = \text{Cons}(\text{shared}(X), q) \\ \text{Cons}(\text{shared}(\{x\}), \psi(v)) \text{ sinon} \end{cases} \right], \rho \rangle \right\}$$

Ensuite, si x : `&mut` τ est déréférencé, une occurence $\text{unique}(x)$ doit se trouver sur la pile d'emprunts correspondant à l'espace mémoire vers lequel il pointe :

$$\mathbb{E}[\![* x]\!]\langle C, \psi, \rho\rangle \triangleq \begin{cases} \{\langle C, \psi[\rho(x) \to \text{until}_{\text{mut}}(x, \psi(\rho(x)))], \rho\rangle\} \text{ si } \text{unique}(x) \in \psi(\rho(x)) \\ \emptyset \text{ sinon} \end{cases}$$

On invalide alors tous les emprunts de la pile situés au dessus de $\text{unique}(x)$, qui ne peuvent désormais plus être utilisés. Par exemple :

```
let mut a = 0;  // (L1)
let b = &mut a; // (L2)
let c = &mut a; // (L3)
*c += 4;        // (L4)
```

Ce code sera validé par le *borrow checker* uniquement dans le cas où c n'est plus utilisé après l'usage de b (L4). Au moment de (L3), on aura $\psi(a) = [\text{unique}(c), \text{unique}(b), \text{unique}(a)]$, puis après l'usage de l'emprunt b (L4), $\psi(a) = [\text{unique}(b), \text{unique}(a)]$.
Si x : `&`τ, on invalide les emprunts uniques qui ont une précédence sur x dans la pile d'emprunts, i.e. les $\text{unique}(t)$, $t \neq x$ et $\text{shared}(X)$, $x \notin X$, d'où :

$$\mathbb{E}[\![* x]\!]\langle C, \psi, \rho\rangle \triangleq \begin{cases} \{\langle C, \psi[\rho(x) \to \text{until}_{\text{shr}}(x, \psi(\rho(x)))], \rho\rangle\} \\ \text{si } \exists X \subseteq \mathcal{V}, \text{shared}(\{x\} \cup X) \in \psi(\rho(x)) \\ \emptyset \text{ sinon} \end{cases}$$

On peut tout de même garder tous les emprunts partagés qui ont été fait au même état mémoire que lors de l'emprunt partagé de x. Par exemple :

`let mut a = 0;  // (L1)`	$\psi(a) = [\text{unique}(a)]$
`let b = &mut a; // (L2)`	$\psi(a) = [\text{unique}(b), \text{unique}(a)]$
`let c = &b;     // (L3)`	$\psi(a) = [\text{shared}(\{c\}), \text{unique}(b), \text{unique}(a)]$
`let d = &b;     // (L4)`	$\psi(a) = [\text{shared}(\{c, d\}), \text{unique}(b), \text{unique}(a)]$
`let e = &mut *b; // (L5)`	$\psi(a) = [\text{unique}(e), \text{unique}(b), \text{unique}(a)]$

$$\mathbb{S}[\![* x = e]\!]\langle C, \psi, \rho\rangle \triangleq \begin{cases} \{\langle C, \psi[\rho(x) \to \text{until}_{\text{mut}}(x, \psi(\rho(x)))], \rho\rangle\} \text{ si } \text{unique}(x) \in \psi(\rho(x)) \\ \emptyset \text{ sinon} \end{cases}$$

De même, le déréférencement d'un emprunt mutable est considéré comme un usage, ainsi $\mathbb{S}[\![* x = e]\!]$ a le même effet sur ψ que $\mathbb{E}[\![* x]\!]$, et il en est de même pour un assignement de $x : \tau$:

$$\mathbb{S}[\![x = e]\!]\langle C, \psi, \rho\rangle \triangleq \begin{cases} \{\langle C, \psi[\rho(x) \to \text{until}_{\text{mut}}(x, \psi(\rho(x)))], \rho\rangle\} \text{ si } \text{unique}(x) \in \psi(\rho(x)) \\ \emptyset \text{ sinon} \end{cases}$$

L'instruction `move` va effectuer une modification en place de la pile d'emprunt pour informellement renommer l'occurence de v en x, tout en prenant en compte que $\texttt{move}(v)$ constitue un usage de v :

$$\mathbb{S}[\![x = \texttt{move}(v)]\!]\langle C, \psi, \rho\rangle \triangleq \begin{cases} \{\langle C, \psi[v \to \text{Cons}(\text{unique}(x), q)], \rho\rangle\} \\ \text{si } v : \tau, v \in \psi(v), \text{until}_{\text{mut}}(v, \psi(v)) = \text{Cons}(\text{unique}(v), q) \\ \{\langle C, \psi[\rho(v) \to \text{Cons}(\text{unique}(x), q)], \rho\rangle\} \\ \text{si } v : \texttt{\&mut } \tau, v \in \psi(\rho(v)), \text{until}_{\text{mut}}(v, \psi(\rho(v))) = \text{Cons}(\text{unique}(v), q) \\ \{\langle C, \psi[\rho(v) \to \text{Cons}(\text{shared}(\{x\}, X), q)], \rho\rangle\} \\ \text{si } v : \texttt{\&}\tau, v \in \psi(\rho(v)), \text{until}_{\text{shr}}(v, \psi(\rho(v))) = \text{Cons}(\text{shared}(\{v\} \cup X), q) \\ \emptyset \text{ sinon} \end{cases}$$

Et enfin, lorsqu'on enlève $x : \tau$ de l'environnement, ψ sera mis à jour pour supprimer la pile d'emprunts correspondant à x et en invalidant tous les éléments de la pile d'emprunts :

$$\mathbb{S}[\![\texttt{storage\_dead}(x)]\!]\langle C, \psi, \rho\rangle \triangleq \langle C, \psi[x \to \bot], \rho[v \to \bot, \forall v \in \psi(x)]\rangle$$

4.2.3. Définition de $\mathcal{E}^\sharp$

Définition 4

De là, on peut définir la sémantique concrète :

$$\mathcal{E}^\sharp \triangleq \bigcup_{C \subseteq \mathcal{V}} \{\langle C, R\rangle \mid R \in \mathcal{P}(C \to \mathbb{Z} \cup \mathcal{P}^{\text{tr}})\}$$

que l'on munit aussi d'un ordre partiel :

$$\langle C, R\rangle \overset{\sharp}{\preccurlyeq} \langle C', R'\rangle \overset{\Delta}{\iff} C' \subseteq C \wedge \left\{\rho_{|C'} \mid \rho \in R\right\} \subseteq R'$$

formant aussi un treillis muni de l'union ensembliste paire-à-paire définie comme suit :

$$\langle C, R\rangle \cup^\sharp \langle C', R'\rangle \triangleq \langle C \cup C', R \cup R'\rangle$$

On va construire une abstraction γ et une concrétisation α :

$$\alpha(X) \triangleq \langle \overline{\text{C}}, \left\{\rho_{|\overline{\text{C}}} \mid \langle C, \rho\rangle \in X\right\}\rangle, \overline{\text{C}} \triangleq \bigcap_{\langle C, \rho\rangle \in X} \{C\}$$

$$\gamma\langle C, R\rangle \triangleq \{\langle C, \rho\rangle \mid \rho \in R\}$$

Théorème

On démontre qu'on a une correspondance Galoisienne entre $\mathcal{E}$ et $\mathcal{E}^\sharp$:

$$(\mathcal{P}(\mathcal{E}), \preccurlyeq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(\mathcal{E}^\sharp, \overset{\sharp}{\preccurlyeq}\right)$$

Cependant, cette sémantique n'est pas calculable car elle est très proche du concret, ce qui motive la suite : pour la rendre calculable on va séparer l'abstraction des variables numériques de celle des références. De cette manière, on pourra abstraire les variables numériques en le domaine abstrait $\mathcal{D}^\sharp$ Section 3, puis garder notre abstraction des références à part.

Définition 5

On définit la sémantique abstraite calculable :

$$\mathcal{E}^\sharp_{\text{cal}} \triangleq \bigcup_{C \subseteq \mathcal{V}} \left\{ \langle C, d^\sharp, \rho^\sharp \rangle \mid \rho^\sharp \in C_{\mathcal{P}^{\text{tr}}} \to \mathcal{P}(\mathcal{P}^{\text{tr}}), d^\sharp \in \mathcal{D}^\sharp(C_{\texttt{int}}) \right\}$$

Avec :

$$C_{\texttt{int}} \triangleq \{c \in C \mid c : \texttt{int}\}$$

$$C_{\mathcal{P}^{\text{tr}}} \triangleq \{c \in C \mid c : \texttt{\&}\tau \vee c : \texttt{\&mut } \tau\}$$

et $\mathcal{D}^\sharp$ un domaine numérique abstrait.

Ce domaine donne une approximation de base des références qui est sûre et calculable, nous indiquant par exemple que dans Section 4.1, on aura :

$$\texttt{mc} \overset{\text{points to}}{\longrightarrow} \{\texttt{a}, \texttt{b}\}$$

mais n'est pas assez précis pour nous indiquer que c pointera vers a seulement sous certaines conditions liées à l'environnement. C'est ce qui motive le rafinement de notre abstraction de base $\mathcal{E}^\sharp$.

4.3. Raffinement $\mathcal{E}^*$

4.3.1. Raffinement $\mathcal{E}^\flat$ de l'état mémoire

Définition 6

Si on note :

$$\mathcal{C}^{\text{ond}} \triangleq \mathcal{P}(\texttt{cond})$$

en notant cond l'ensemble des conditions induit de la grammaire Section 1.1, et que l'on munit $\mathcal{C}^{\text{ond}}$ d'un ordre partiel pour $c, c' \in \mathcal{C}^{\text{ond}}$:

$$c \sqsubseteq c' \overset{\Delta}{\iff} c' \subseteq c$$

et d'une union :

$$c \sqcup c' \triangleq c \cap c'$$

$\left(\mathcal{C}^{\text{ond}}, \sqsubseteq, \sqcup\right)$ forme un treillis qui définit un état abstrait des hypothèses.

Intuitivement, l'ordre $\sqsubseteq$ indique qu'un état abstrait ayant plus d'hypothèses sera plus précis.

Exemple

Si on prend le programme suivant :

```
if (*a >= *b) { if (*c <= *d) {
    x = 2 // (L2)
} }
```

On aura $c \in \mathcal{C}^{\text{ond}} = \{(\texttt{*a >= *b}), (\texttt{*c <= *d})\}$ au moment de (L2) dont la conjonction correspond bien aux hypothèses faites dans ce chemin d'exécution.

A partir de cet état abstrait des hypothèses, on aimerait garder l'information des hypothèses associées aux emprunts, de manière à avoir plus de précision sur quel emprunt correspond à quel hypothèse, comme dans le cas de Section 4.1, où l'on voudrait garder :

$$\texttt{mc} \to \{(\texttt{a}, \texttt{*ma >= *mb}), (\texttt{b}, \texttt{*ma < *mb})\}$$

Donc informellement, savoir que `mc` pointe sur `a` si `*ma >= *mb`, et pointe sur `b` si `*ma < *mb`.

Définition 7

On définit l'état mémoire concret instrumenté $\left(\mathcal{P}(\mathcal{E}^\flat), \subseteq, \cup\right)$ avec :

$$\mathcal{E}^\flat \triangleq \bigcup_{C \subseteq \mathcal{V}} \{\langle C, \rho\rangle \mid \rho \in C \to \mathbb{Z} \cup \left(\mathcal{P}^{\text{tr}} \times \mathcal{C}^{\text{ond}}\right)\}$$

Cet état se souvient, avec chaque environnement, des contraintes accumulées lors de l'exécution jusqu'à ce point.

4.3.2. Abstraction $\mathcal{E}^\sharp$ de l'état mémoire

Définition 8

On raffine notre abstraction $\mathcal{E}^\sharp$ en $(\mathcal{E}^\sharp, \subseteq, \cup)$ avec :

$$\mathcal{E}^\sharp \triangleq \bigcup_{C \subseteq \mathcal{V}} \left\{\langle C, d^\sharp, \rho^\sharp\rangle \mid \rho^\sharp \in \mathcal{P}_{\text{finite}}\left((C_{\mathcal{P}^{\text{tr}}} \to \mathcal{P}(\mathcal{P}^{\text{tr}})) \times \mathcal{C}^{\text{ond}}\right), d^\sharp \in \mathcal{D}^\sharp(C_{\texttt{int}})\right\}$$

Ce modèle est inspiré de la modélisation des références induite dans le modèle de *RustBelt* [3]. Informellement, $\rho(v) \in \mathcal{P}\left(\mathcal{P}^{\text{tr}} \times \mathcal{C}^{\text{ond}}\right)$[1] représente une surapproximation des valeurs vers lesquelles $v \in \mathcal{V}_{\&\tau}$ peut pointer sous condition, ie. $(x, c) \in \rho(v)$ indique :

$$v \xrightarrow{\texttt{assume}(c)} x$$

Par exemple, dans Section 4.1, on aurait donc :

$$\texttt{mc} \xrightarrow{\texttt{assume(*a>=*b)}} \texttt{a}$$

$$\texttt{mc} \xrightarrow{\texttt{assume(*a<*b)}} \texttt{b}$$

Cette abstraction nous permet de retenir plus d'informations sur l'état mémoire, tout en restant sûr et calculable. On en déduit une sémantique transitionnelle pour l'état abstrait $\langle C, d^\sharp, \rho^\sharp\rangle$, avec x : `&mut` τ, x : `&`τ :

$$\mathbb{S}^\sharp[\![\texttt{storage\_live}(x)]\!]\langle C, d^\sharp, \rho^\sharp\rangle \triangleq \langle C, d^\sharp, \{(\rho[x \to \top], c) \mid (\rho, c) \in \rho^\sharp\}\rangle$$

$$\mathbb{S}^\sharp[\![\texttt{storage\_dead}(x)]\!]\langle C, d^\sharp, \rho^\sharp\rangle \triangleq \langle C, d^\sharp, \{(\rho[x \to \bot], c) \mid (\rho, c) \in \rho^\sharp\}\rangle$$

$$\mathbb{S}^\sharp[\![t = \texttt{\&mut } x]\!]\langle C, d^\sharp, \rho^\sharp\rangle \triangleq \langle C, d^\sharp, \{(\rho[t \to \{x\}], c) \mid (\rho, c) \in \rho^\sharp\}\rangle$$

$$\mathbb{S}^\sharp[\![t = \texttt{\&}x]\!]\langle C, d^\sharp, \rho^\sharp\rangle \triangleq \langle C, d^\sharp, \{(\rho[t \to \{x\}], c) \mid (\rho, c) \in \rho^\sharp\}\rangle$$

$$\mathbb{S}^\sharp[\![x = \texttt{move}(v)]\!]\langle C, d^\sharp, \rho^\sharp\rangle \triangleq \langle C, d^\sharp, \{(\rho[x \to \rho(v), v \to \{\text{INVALID}\}], c) \mid (\rho, c) \in \rho^\sharp\}\rangle$$

Pour le `move`, on prend soin d'invalider v après la transmission des permissions à x.

[1] On note par abus $\rho(v) = \bigcup_{\rho_i \in \rho^\sharp} \rho_i(v) \in \mathcal{P}\left(\mathcal{P}^{\text{tr}} \times \mathcal{C}^{\text{ond}}\right)$

$$\mathbb{S}^\sharp[\![\text{if}(c)\{s_1\} \text{ else } \{s_2\}]\!]\langle C, d^\sharp, \rho^\sharp\rangle \triangleq$$

$$\texttt{let } \langle C_1, d_1^\sharp, \rho_1^\sharp\rangle = \left(\mathbb{S}^\sharp[\![s_1]\!] \circ \mathbb{C}^\sharp[\![c]\!]\right)\langle C, d^\sharp, \rho^\sharp\rangle \texttt{ in}$$

$$\texttt{let } \langle C_2, d_2^\sharp, \rho_2^\sharp\rangle = \left(\mathbb{S}^\sharp[\![s_2]\!] \circ \mathbb{C}^\sharp[\![\neg c]\!]\right)\langle C, d^\sharp, \rho^\sharp\rangle \texttt{ in}$$

$$\langle C_1 \cup C_2, d_1^\sharp \cup^\sharp d_2^\sharp, \left\{(\rho, \text{cond} \cup \{c\}) \mid (\rho, \text{cond}) \in \rho_1^\sharp\right\} \cup \left\{(\rho, \text{cond} \cup \{\neg c\}) \mid (\rho, \text{cond}) \in \rho_2^\sharp\right\}\rangle$$

On accumule dans chaque sous environnement les conditions correspondant aux branches d'exécution prises. Et enfin, dans le cas où x : `&mut` τ :

$$\mathbb{S}^\sharp[\![* x = e]\!]\langle C, d^\sharp, \rho^\sharp\rangle \triangleq \bigcup_{(\rho,c)\in\rho^\sharp} \left(\bigcup_{v\in\rho(x)} \left(\mathbb{S}^\sharp[\![v = e]\!] \circ \mathbb{C}^\sharp[\![c]\!]\right)\langle C, d^\sharp, \{\rho\}\rangle \right)$$

informellement, dans chaque environnement (ρ, c), on effectue l'assignement sous la condition c des valeurs possibles vers lesquelles x pointe.

5. Propriétés d'intérêt particulières au Rust

Comme vu précédemment (Section 4.2.2), le *borrow checker* nous assure que seulement une variable a la propriété d'un espace mémoire. Afin de gagner en expressivité, on autorise des transferts temporaires et définitifs d'appartenance. Ces transferts sont modélisés en pratique par :

$$x = \texttt{move}(v)$$

$$x = \texttt{\&mut } v$$

qui sont deux composants essentiels du langage. Seule la variable ayant l'appartenance (temporairement ou définitivement) de l'espace mémoire est en capacité de l'utiliser de manière mutable. Par exemple :

Programme λ_{MIR}	Programme Rust
`storage_live(x : int);`	`let mut x = 4;`
`storage_live(t : &mut int);`	`let t = &mut x;`
`x = 4;`	`x = 5; // (L5)`
`t = &mut x;`	`[...] // usage de t`
`x = 5; // (L5)`	
`[...] // usage de t`	

Ce programme n'est pas valide car au moment de (L5), t est le garant de l'espace mémoire correspondant à x. En pratique, le compilateur vérifie au moment du *borrow checking* ces emprunts et ne compilera effectivement pas le programme si ils se trouvent être invalides.

Définition 9

Pour vérifier que le *borrow checking* est en effet correct, on ajoute à l'état abstrait $\langle C, d^\sharp, \rho^\sharp \rangle$ une dernière composante $\psi^\sharp : C_{\texttt{int}} \to \mathcal{P}(\mathrm{List}(I_C))$ qui représente une abstraction de ψ Section 4.2.2. On a alors :

$$\mathcal{E}^\star_{\text{chk}} \triangleq \bigcup_{C \subseteq \mathcal{V}} \left\{ \langle C, d^\sharp, \rho^\sharp, \psi^\sharp \rangle \mid \langle C, d^\sharp, \rho^\sharp \rangle \in \mathcal{E}^\star_C, \psi^\sharp \in \mathcal{P}_{\text{finite}}(C_{\texttt{int}} \to \mathcal{P}(\mathrm{List}(I_C))) \right\}$$

6. *Raw pointeurs*

Pour garder une expréssivité similaire au C, dans des cas où la discipline imposée par le *borrow checker* s'avère trop stricte, une option peut être d'utiliser des *raw pointeurs*. Les *raw pointeurs* du Rust sont similaires aux pointeurs du C, ils ne sont pas vérifiés par le compilateur et n'ont pas de restrictions d'alias. Par exemple :

```
let mut x = 0;                    // (L1)
let ptr1 = &mut x as *mut i32;    // (L2)
let ptr2 = ptr1;                  // (L3)
unsafe { *ptr1 = 1; }             // (L4)
unsafe { println!("{}", *ptr2); } // (L5) affiche 1
```

A la ligne (L2), on va convertir le type `&mut i32` en `*mut i32`. Ce *raw pointeur* peut ensuite être dupliqué comme à la ligne (L3). La création de *raw pointeurs* fait partie du langage sûr, mais lorsque l'on veut accéder au contenu du *raw pointeur* en le déréférençant pour y lire ou y écrire, l'opération est marquée `unsafe`, indiquant que c'est à l'appréciation de l'utilisateur de s'assurer de la sûreté de ses opérations. Informellement, pour s'assurer de leur sûreté le programmeur doit vérifier que l'espace mémoire pointé est valide au moment du déréférencement.

6.1. Grammaire et typage

On ajoute à la grammaire les opérations de conversion de type ainsi que les types associés aux *raw pointeurs* :

Instruction	s	+=	$v = v$ as *mut τ	*conversion mutable*
		|	$v = v$ as *const τ	*conversion immuable*
Type	τ	+=	*const τ	*raw pointeur immuable*
		|	*mut τ	*raw pointeur mutable*

Et on se munit à nouveau d'un jugement de type $\Gamma : \mathcal{V} \to \mathrm{T}$:

$$\frac{\Gamma, x : \texttt{\&mut } \tau \vdash v : \texttt{*mut } \tau}{\Gamma, x : \texttt{\&mut } \tau \vdash (v = x \text{ as } \texttt{*mut } \tau) : \texttt{ok}} (S_{11})$$

$$\frac{\Gamma, x : \&\tau \vdash v : \texttt{*const } \tau}{\Gamma, x : \&\tau \vdash (v = x \text{ as } \texttt{*const } \tau) : \texttt{ok}} (S_{12}) \qquad \frac{\Gamma, v : \tau \vdash x : \texttt{*mut } \tau}{\Gamma, v : \tau \vdash (*x = v) : \texttt{ok}} (S_{13})$$

$$\frac{\Gamma, e_1 : \texttt{*mut } \tau \vdash e_2 : \texttt{*mut } \tau}{\Gamma, e_1 : \texttt{*mut } \tau \vdash (e_1 = e_2) : \texttt{ok}}(S_1) \qquad \frac{\Gamma, e_1 : \texttt{*const } \tau \vdash e_2 : \texttt{*const } \tau}{\Gamma, e_1 : \texttt{*const } \tau \vdash (e_1 = e_2) : \texttt{ok}}(S_1)$$

$$\frac{\Gamma \vdash x : \texttt{*const } \tau}{\Gamma \vdash (*x) : \tau}(T_7) \qquad \frac{\Gamma \vdash x : \texttt{*mut } \tau}{\Gamma \vdash (*x) : \tau}(T_8)$$

Ainsi, on garde le même opérateur `*` pour le déréférencement. Simplement, sa sémantique se verra transformée dans le cas des *raw pointeurs* ci-dessous. On remarque aussi que les *raw pointeurs* ont une sémantique associée à l'assignation contrairement aux références, car les alias sont autorisés.

6.2. Sémantique transitionnelle des programmes

On garde le même état d'un programme Rust :

$$\mathcal{S} \triangleq \mathcal{V} \to \mathbb{Z} \cup \mathcal{P}^{\text{tr}}$$

car cet état mémoire englobe déjà les valeurs d'un pointeur, qui sont les mêmes qu'une référence. On en déduit une sémantique transitionnelle pour un ensemble d'états Σ :

$$\mathbb{S}[\![\texttt{storage\_live}(v : \texttt{*mut } \tau)]\!]\Sigma \triangleq \{\sigma[v \to \text{UNINIT}] \mid \sigma \in \Sigma\}$$

$$\mathbb{S}[\![\texttt{storage\_live}(v : \texttt{*const } \tau)]\!]\Sigma \triangleq \{\sigma[v \to \text{UNINIT}] \mid \sigma \in \Sigma\}$$

$$\mathbb{S}[\![\texttt{storage\_dead}(v : \texttt{*mut } \tau)]\!]\Sigma \triangleq \{\sigma[v \to \text{INVALID}, \forall x \in \mathcal{V}, \text{tq. } \sigma(x) = v, x \to \text{INVALID}] \mid \sigma \in \Sigma\}$$

$$\mathbb{S}[\![\texttt{storage\_dead}(v : \texttt{*const } \tau)]\!]\Sigma \triangleq \{\sigma[v \to \text{INVALID}, \forall x \in \mathcal{V}, \text{tq. } \sigma(x) = v, x \to \text{INVALID}] \mid \sigma \in \Sigma\}$$

Les *raw pointeurs* agissent donc à l'allocation et la désallocation comme des références. Pour x, v : `*const` τ ou `*mut` τ :

$$\mathbb{S}[\![x = v]\!]\Sigma \triangleq \{\sigma[x \to u] \mid u \in \mathbb{E}[\![v]\!]\sigma, \sigma \in \Sigma\}$$

On autorise la création d'alias par l'assignation directe, contrairement aux références où le seul moyen de transmettre les valeurs d'une référence à une autre s'opère par l'instruction `move`, qui **invalide** l'ancienne variable. Puis pour y : `*mut` τ :

$$\mathbb{S}[\![* y = e]\!]\Sigma \triangleq \{\sigma[x \to v] \mid x \in \mathbb{E}[\![* y]\!]\sigma, v \in \mathbb{E}[\![e]\!]\sigma, \sigma \in \Sigma\}$$

Enfin, pour x : `*const` τ, x : `*mut` τ et σ état d'entrée :

$$\mathbb{E}[\![* x]\!]\sigma \triangleq \bigcup_{\substack{u \in \sigma(x) \\ u \neq \text{INVALID} \\ u \neq \text{UNINIT}}} \{\sigma(u)\}$$

6.3. Sémantique abstraite du *Borrow Checker* pour les *raw pointeurs*

On aimerait un modèle permettant de s'assurer que les garanties mémoires du Rust restent inviolées en dehors du cadre de l'exécution de blocs marqués `unsafe`. Pour cela, il nous faut l'information d'un emprunt par un *raw pointeur* dans la pile d'emprunt correspondant à l'espace mémoire. On ajoute donc à $I_{\mathcal{V}}$ un objet sharedRW qui modélise l'emprunt par des *raw pointeurs*, i.e. :

$$I_{\mathcal{V}} \triangleq I_{\mathcal{V}} \cup \{\text{sharedRW}\}$$

qui modélise informellement l'information d'un emprunt par un *raw pointeur*. On définit un opérateur $\text{until}_{\text{raw}}$ pour $v \in \mathcal{V}$ de la même manière que dans Section 4.2.2 :

$$\text{until}_{\text{raw}}(\psi(v)) \triangleq \begin{cases} \psi(v) \text{ si } \psi(v) = \text{Cons}(\text{sharedRW}, q) \\ \text{until}_{\text{raw}}(q) \text{ si } \psi(v) = \text{Cons}(t, q), t \neq \text{sharedRW} \\ \text{Nil si } \psi(v) = \text{Nil} \end{cases}$$

Avec la sémantique transitionnelle suivante pour l'état concret $\langle C, \psi, \rho \rangle$:

$$\mathbb{S}[\![x = v \text{ as } \texttt{*mut } \tau]\!]\langle C, \psi, \rho \rangle \triangleq \begin{cases} \{\langle C, \psi[\rho(v) \to \text{Cons}(\text{sharedRW}, \text{until}_{\text{mut}}(v, \psi(\rho(v))))], \rho\rangle\} \\ \text{si unique}(v) \in \psi(\rho(v)) \\ \emptyset \text{ sinon} \end{cases}$$

$$\mathbb{S}[\![x = v \text{ as } \texttt{*const } \tau]\!]\langle C, \psi, \rho \rangle \triangleq \begin{cases} \{\langle C, \psi[\rho(v) \to \text{Cons}(\text{sharedRW}, \text{until}_{\text{shr}}(v, \psi(\rho(v))))], \rho\rangle\} \\ \text{si } \exists X \subseteq \mathcal{V}, \text{shared}(\{v\} \cup X) \in \psi(\rho(v)) \\ \emptyset \text{ sinon} \end{cases}$$

Ainsi, la création d'un *raw pointeur* x est considérée comme un usage de la référence v qu'il a transtypée. Puis, pour x : `*mut` τ, `*const` τ :

$$\mathbb{E}[\![* x]\!]\langle C, \psi, \rho \rangle \triangleq \begin{cases} \{\langle C, \psi[\rho(x) \to \text{until}_{\text{raw}}(\psi(\rho(x)))], \rho\rangle\} \text{ si sharedRW} \in \psi(\rho(x)) \\ \emptyset \text{ sinon} \end{cases}$$

$$\mathbb{S}[\![* x = e]\!]\langle C, \psi, \rho \rangle \triangleq \begin{cases} \{\langle C, \psi[\rho(x) \to \text{until}_{\text{raw}}(\psi(\rho(x)))], \rho\rangle\} \text{ si sharedRW} \in \psi(\rho(x)) \\ \emptyset \text{ sinon} \end{cases}$$

Par exemple :

```
let mut a = 0;                  // (L1)
let b = &mut a;                 // (L2)
let c = &mut *b as *mut i32;    // (L3)
*b = 4;                         // (L4)
unsafe { *c += 2; }             // (L5)
```

$$\psi(a) = [\text{unique}(a)]$$
$$\psi(a) = [\text{unique}(b), \text{unique}(a)]$$
$$\psi(a) = [\text{sharedRW}, \text{unique}(b), \text{unique}(a)]$$
$$\psi(a) = [\text{unique}(b), \text{unique}(a)]$$
$$\psi(a) = \emptyset$$

On aura $\psi(a) = \emptyset$ à (L5) bien que un programme Rust accepte ce code car il brise l'invariant d'un environnement à un seul possesseur à la fois pour a : ici b et c sont possesseurs en simultané car ils sont utilisés à la suite après leur définition, qui est contraire à la philosophie du langage.

Bibliographie

[1] A. M. David Delmas Abdelraouf Ouadjaout, “Static Analysis of Endian Portability by Abstract Interpretation,” *HAL*, 2021, doi: https://doi.org/10.5281/zenodo.5206794.

[2] J. K. D. D. Ralf Jung Hoang-Hai Dang, “Stacked borrows: an aliasing model for Rust,” *ACM*, 2020, doi: https://doi.org/10.5281/zenodo.3541779.

[3] R. K. D. D. Ralf Jung Jacques-Henri Jourdan, “RustBelt: securing the foundations of the Rust programming language,” *ACM*, 2017, doi: https://doi.org/10.1145/3158154.