

Remonter les barrières pour ouvrir une clôture

Paul Patault Arnaud Golfouse Xavier Denis

Janvier 2025 @ JFLA

université
PARIS-SACLAY

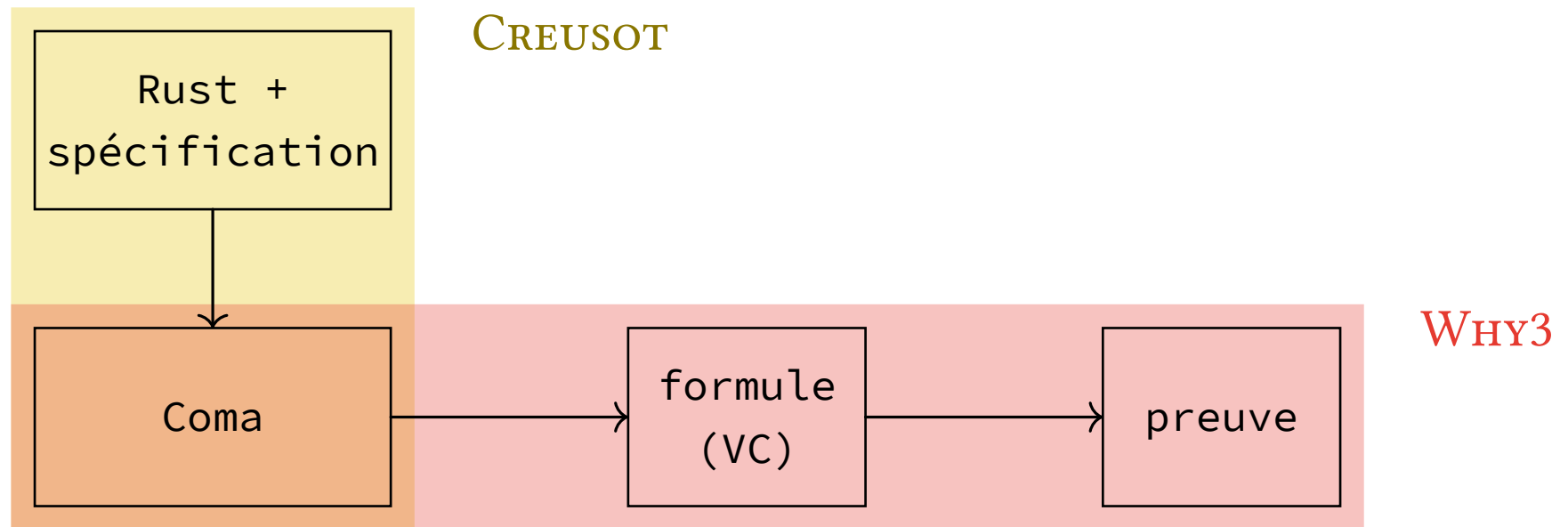
Inria



Laboratoire
Méthodes
Formelles

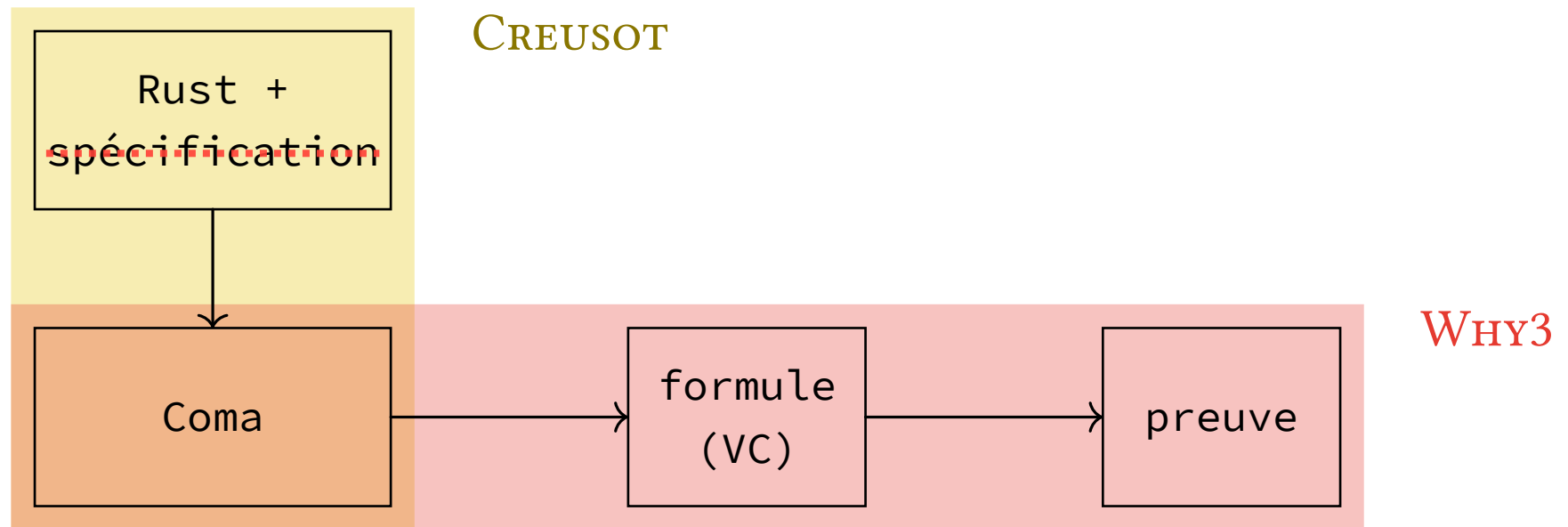
CREUSOT

Outil de vérification déductive pour Rust, basé sur WHY3



CREUSOT

Outil de vérification déductive pour Rust, basé sur WHY3



CREUSOT

```
fn map<F: Fn(i32) -> i32>(opt: Option<i32>, f: F)
  -> Option<i32> {
    match opt {
      None      => None,
      Some(x)   => Some(f(x)),
    }
  }
```

CREUSOT

```
#[requires( $\forall x. \text{opt} = \text{Some}(x) \Rightarrow \text{f.pre}(x)$ )]  
#[ensures( $\forall x. \text{opt} = \text{Some}(x) \Rightarrow$   
     $\exists y. \text{result} = \text{Some}(y) \wedge \text{f.post}(x, y)$ )]  
fn map<F: Fn(i32) -> i32>(opt: Option<i32>, f: F)  
    -> Option<i32> {  
    match opt {  
        None      => None,  
        Some(x)   => Some(f(x)),  
    }  
}
```

CREUSOT

```
#[requires( $\forall x. \text{opt} = \text{Some}(x) \Rightarrow \text{f.pre}(x)$ )]  
#[ensures( $\forall x. \text{opt} = \text{Some}(x) \Rightarrow$   
     $\exists y. \text{result} = \text{Some}(y) \wedge \text{f.post}(x, y)$ )]  
fn map<F: Fn(i32) -> i32>(opt: Option<i32>, f: F)  
    -> Option<i32> {  
    match opt {  
        None      => None,  
        Some(x)   => Some(f(x)),  
    }  
}  
  
let o = Some(21);  
let double = |x| x * 2;  
assert!(map(o, double).unwrap() == 42);
```

Problème

Il faut écrire la spécification de la clôture !

```
let double =  
  #[requires(x * 2 ≤ i32::MAX)]  
  #[ensures(result = x * 2)]  
  |x| x * 2;
```

C'est très lourd...

Problème

Il faut écrire la spécification de la clôture !

```
let double =  
  #[requires(x * 2 ≤ i32::MAX)]  
  #[requires(x * 2 ≥ i32::MIN)]  
  #[ensures(result = x * 2)]  
  |x| x * 2;
```

C'est très lourd... et on se trompe facilement !

Gestion des clôtures

La spécification de map est monorphisée :

```
#[requires( $\forall x. \text{opt} = \text{Some}(x) \Rightarrow$   
            $i32::\text{MIN} \leq x * 2 \leq i32::\text{MAX})]$   
#[ensures( $\forall x. \text{opt} = \text{Some}(x) \Rightarrow$   
           $\exists y. \text{result} = \text{Some}(y) \wedge y = x * 2)$ ]  
fn map_double(opt: Option<i32>) -> Option<i32> {  
    ...  
}
```

On utilise directement la spécification donnée à $|x| \leq x * 2$.

Ce qu'on veut

- ne pas écrire la spécification de la clôture
↳ « la spécification est le code »
- conserver les prédicats $f.pre$ et $f.post$ pour monorphiser !

Ce qu'on veut

- ne pas écrire la spécification de la clôture
↳ « la spécification est le code »
- conserver les prédicats $f.pre$ et $f.post$ pour monorphiser !

COMA

INTERLUDE

COMA

```
let double (x: i32) (ret (r: i32)) =
```

```
    ret (x * 2)
```

COMA

```
let double (x: i32) (ret (r: i32)) =  
  assert {  
    i32_min ≤ x * 2 ≤ i32_max  
  } ↑ ret (x * 2)
```

COMA

```
let double (x: i32) (ret (r: i32)) =  
  assert {  
    i32_min ≤ x * 2 ≤ i32_max  
  } ↑ out (x * 2)  
/ out (result: i32) =  
  
  ret result
```

CoMA

```
let double (x: i32) (ret (r: i32)) =  
  assert {  
    i32_min ≤ x * 2 ≤ i32_max  
  } ↑ out (x * 2)  
/ out (result: i32) =  
  assert { result = x * 2 }  
  ↑ ret result
```

VCgen de COMA (simplifié)

Générateur modal de conditions de vérification

mode définition

$$\mathcal{D}(\uparrow e) \triangleq \mathcal{D}(e) \wedge \mathcal{A}(e)$$

$$\mathcal{D}(\{\varphi\} e) \triangleq \varphi \rightarrow \mathcal{D}(e)$$

$$\mathcal{D}(h \bar{a}) \triangleq h^{\natural} \bar{a}$$

mode appelant

$$\mathcal{A}(\uparrow e) \triangleq \top$$

$$\mathcal{A}(\{\varphi\} e) \triangleq \varphi \wedge (\varphi \rightarrow \mathcal{A}(e))$$

$$\mathcal{A}(h \bar{a}) \triangleq h \bar{a}$$

VCgen de COMA (simplifié)

Générateur modal de conditions de vérification

mode définition

$$\mathcal{D}(\uparrow e) \triangleq \mathcal{D}(e) \wedge \mathcal{A}(e)$$

$$\mathcal{D}(\{\varphi\} e) \triangleq \varphi \rightarrow \mathcal{D}(e)$$

$$\mathcal{D}(h \bar{a}) \triangleq h^{\sharp} \bar{a}$$

mode appelant

$$\mathcal{A}(\uparrow e) \triangleq \top$$

$$\mathcal{A}(\{\varphi\} e) \triangleq \varphi \wedge (\varphi \rightarrow \mathcal{A}(e))$$

$$\mathcal{A}(h \bar{a}) \triangleq h \bar{a}$$

$$\hat{\mathcal{A}}(\text{let } h \bar{a} = e) \triangleq \lambda \bar{a}. \mathcal{A}(e)$$

VCgen de COMA (simplifié)

Générateur modal de conditions de vérification

mode définition

$$\mathcal{D}(\uparrow e) \triangleq \mathcal{D}(e) \wedge \mathcal{A}(e)$$

$$\mathcal{D}(\{\varphi\} e) \triangleq \varphi \rightarrow \mathcal{D}(e)$$

$$\mathcal{D}(h \bar{a}) \triangleq h^{\sharp} \bar{a}$$

mode appelant

$$\mathcal{A}(\uparrow e) \triangleq \top$$

$$\mathcal{A}(\{\varphi\} e) \triangleq \varphi \wedge (\varphi \rightarrow \mathcal{A}(e))$$

$$\mathcal{A}(h \bar{a}) \triangleq h \bar{a}$$

$$\hat{\mathcal{A}}(\text{let } h \bar{a} = e) \triangleq \lambda \bar{a}. \mathcal{A}(e)$$

double ... sans barrières ? oui !

```
let double (x: i32) (ret (r: i32)) =  
  ret (x * 2)
```

pré- et postcondition

$$\hat{\mathcal{A}}(\text{let double } x \text{ } ret = e) = \lambda x : \text{i32}. \lambda ret : \text{i32} \rightarrow \text{Prop}. \\ (\text{i32_min} \leq x \times 2 \leq \text{i32_max}) \wedge \\ ret (x \times 2)$$

pré- et postcondition

$$\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) = \lambda x : \text{i32}. \lambda \text{ret} : \text{i32} \rightarrow \text{Prop}.$$

$$(\text{i32_min} \leq x \times 2 \leq \text{i32_max}) \wedge \\ \text{ret } (x \times 2)$$

$$\text{double'pre} = \lambda x : \text{i32}. \hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } _ \mapsto \top)$$

pré- et postcondition

$$\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) = \lambda x : \text{i32}. \lambda \text{ret} : \text{i32} \rightarrow \text{Prop}.$$

$$(\text{i32_min} \leq x \times 2 \leq \text{i32_max}) \wedge \\ \text{ret } (x \times 2)$$

$$\begin{aligned} \text{double'pre} &= \lambda x : \text{i32}. \hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } _ \mapsto \top) \\ &= \lambda x : \text{i32}. \text{i32_min} \leq x \times 2 \leq \text{i32_max} \end{aligned}$$

pré- et postcondition

$$\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) = \lambda x : \text{i32}. \lambda \text{ret} : \text{i32} \rightarrow \text{Prop}.$$

$$(\text{i32_min} \leq x \times 2 \leq \text{i32_max}) \wedge$$

$$\text{ret } (x \times 2)$$

WP(e, T)

$$\text{double'pre} = \lambda x : \text{i32}. \hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } _ \mapsto \text{T})$$

$$= \lambda x : \text{i32}. \text{i32_min} \leq x \times 2 \leq \text{i32_max}$$

pré- et postcondition

$$\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) = \lambda x : \text{i32}. \lambda \text{ret} : \text{i32} \rightarrow \text{Prop}.$$

$$(\text{i32_min} \leq x \times 2 \leq \text{i32_max}) \wedge$$

$$\text{ret } (x \times 2)$$

WP(e, T)

$$\text{double'pre} = \lambda x : \text{i32}. \hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } _ \mapsto \top)$$

$$= \lambda x : \text{i32}. \text{i32_min} \leq x \times 2 \leq \text{i32_max}$$

$$\text{double'post} = \lambda x, r : \text{i32}.$$

$$\neg(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } y \mapsto r \neq y))$$

let : opérateur de « neutralisation »

pré- et postcondition

$$\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) = \lambda x : \text{i32}. \lambda \text{ret} : \text{i32} \rightarrow \text{Prop}.$$

$$(\text{i32_min} \leq x \times 2 \leq \text{i32_max}) \wedge \text{ret } (x \times 2)$$

WP(e, T)

$$\begin{aligned} \text{double'pre} &= \lambda x : \text{i32}. \hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } _ \mapsto \top) \\ &= \lambda x : \text{i32}. \text{i32_min} \leq x \times 2 \leq \text{i32_max} \end{aligned}$$

$$\text{double'post} = \lambda x, r : \text{i32}.$$

$$\neg(\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } y \mapsto r \neq y))$$

$\hat{\mathcal{A}}$: opérateur de « neutralisation »

pré- et postcondition

$$\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) = \lambda x : \text{i32}. \lambda \text{ret} : \text{i32} \rightarrow \text{Prop}.$$

$$(\text{i32_min} \leq x \times 2 \leq \text{i32_max}) \wedge \text{ret } (x \times 2)$$

WP(e, T)

$$\begin{aligned} \text{double'pre} &= \lambda x : \text{i32}. \hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } _ \mapsto \top) \\ &= \lambda x : \text{i32}. \text{i32_min} \leq x \times 2 \leq \text{i32_max} \end{aligned}$$

$$\text{double'post} = \lambda x, r : \text{i32}.$$

$$\begin{aligned} &\neg(\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } y \mapsto r \neq y)) \\ &= \lambda x, r : \text{i32}. r = x \times 2 \end{aligned}$$

$\hat{\mathcal{A}}$: opérateur de « neutralisation »

pré- et postcondition

$$\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) = \lambda x : \text{i32}. \lambda \text{ret} : \text{i32} \rightarrow \text{Prop}.$$

$$(\text{i32_min} \leq x \times 2 \leq \text{i32_max}) \wedge \text{ret } (x \times 2)$$

WP(e, T)

$$\text{double'pre} = \lambda x : \text{i32}. \hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } _ \mapsto \text{T})$$

$$= \lambda x : \text{i32}. \text{i32_min} \leq x \times 2 \leq \text{i32_max}$$

$$\text{double'post} = \lambda x, r : \text{i32}.$$

$$\neg(\hat{\mathcal{A}}(\text{let double } x \text{ ret} = e) \ x \ (\text{fun } y \mapsto r \neq y))$$

$$= \lambda x, r : \text{i32}. r = x \times 2$$

SP(e, WP(e, T))

$\hat{\mathcal{A}}$: opérateur de « neutralisation »

Évaluation

exemple	sans ↑	LoC	LoS	temps (s)
bool_then	X	24	10	0,83
	✓	18	8	0,83
option	X	52	31	0,97
	✓	24	12	0,89
iterator	X	42	15	2,79
	✓	24	9	2,50
avl	X	105	155	1,34
	✓	101	99	1,50

```

#[requires( $\forall x. \text{opt} = \text{Some}(x) \implies \text{f.pre}(x)$ )]
#[ensures( $\forall x. \text{opt} = \text{Some}(x) \implies$ 
     $\exists y. \text{result} = \text{Some}(y) \wedge \text{f.post}(x, y)$ )]
fn map<F: Fn(i32) -> i32>(opt: Option<i32>, f: F)
    -> Option<i32> {
    match opt {
        None      => None,
        Some(x)   => Some(f(x)),
    }
}

let o = Some(21);
let double =
    #[requires( $x * 2 \leq \text{i32::MAX}$ )]
    #[requires( $x * 2 \geq \text{i32::MIN}$ )]
    #[ensures( $\text{result} = x * 2$ )]
    |x| x * 2;
assert!(map(o, double).unwrap() == 42);

```

```

#[requires( $\forall x. \text{opt} = \text{Some}(x) \implies \text{f.pre}(x)$ )]
#[ensures( $\forall x. \text{opt} = \text{Some}(x) \implies$ 
     $\exists y. \text{result} = \text{Some}(y) \wedge \text{f.post}(x, y)$ )]
fn map<F: Fn(i32) -> i32>(opt: Option<i32>, f: F)
    -> Option<i32> {
    match opt {
        None      => None,
        Some(x)   => Some(f(x)),
    }
}

let o = Some(21);
let double = |x| x * 2;
assert!(map(o, double).unwrap() == 42);

```

Ce qu'on a obtenu

- ✓ ne pas écrire la spécification de la clôture
- ✓ conserver les prédicats $f.pre$ et $f.post$ pour monorphiser

Merci

Paul, Arnaud, Xavier