Remonter les barrières pour ouvrir une clôture

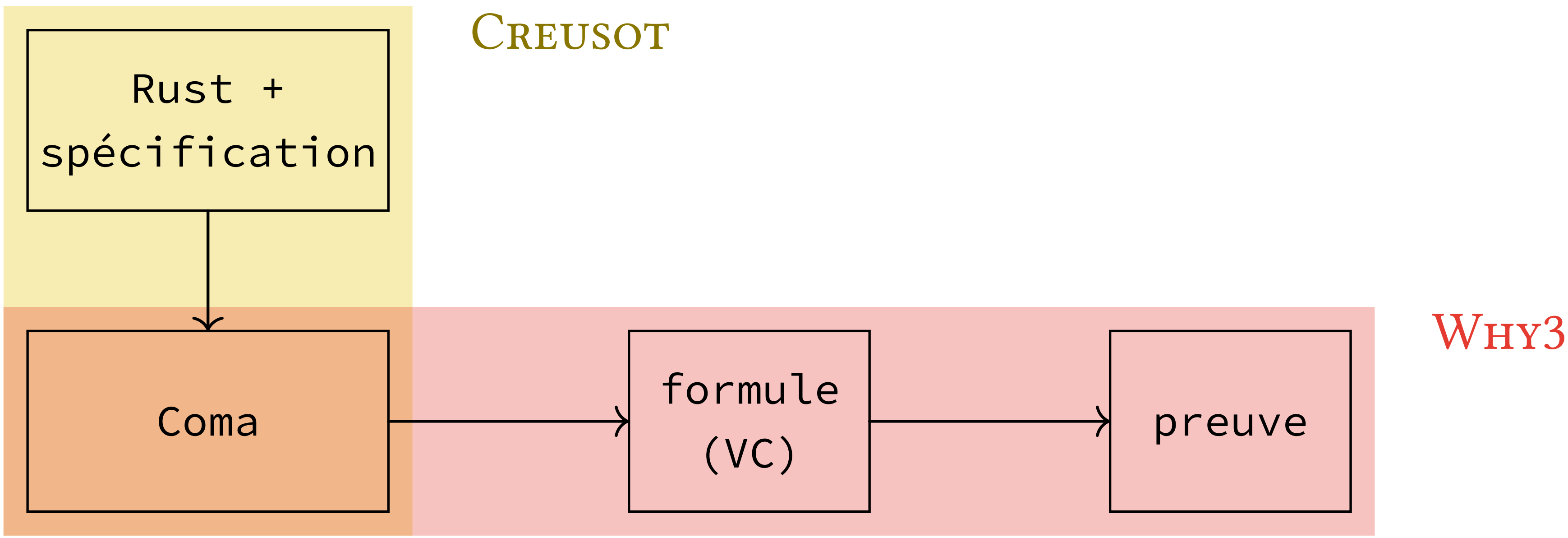
Paul Patault   Arnaud Golfouse   Xavier Denis

Janvier 2025 @ JFLA

Université Paris-Saclay   Inria   LMF Laboratoire Méthodes Formelles

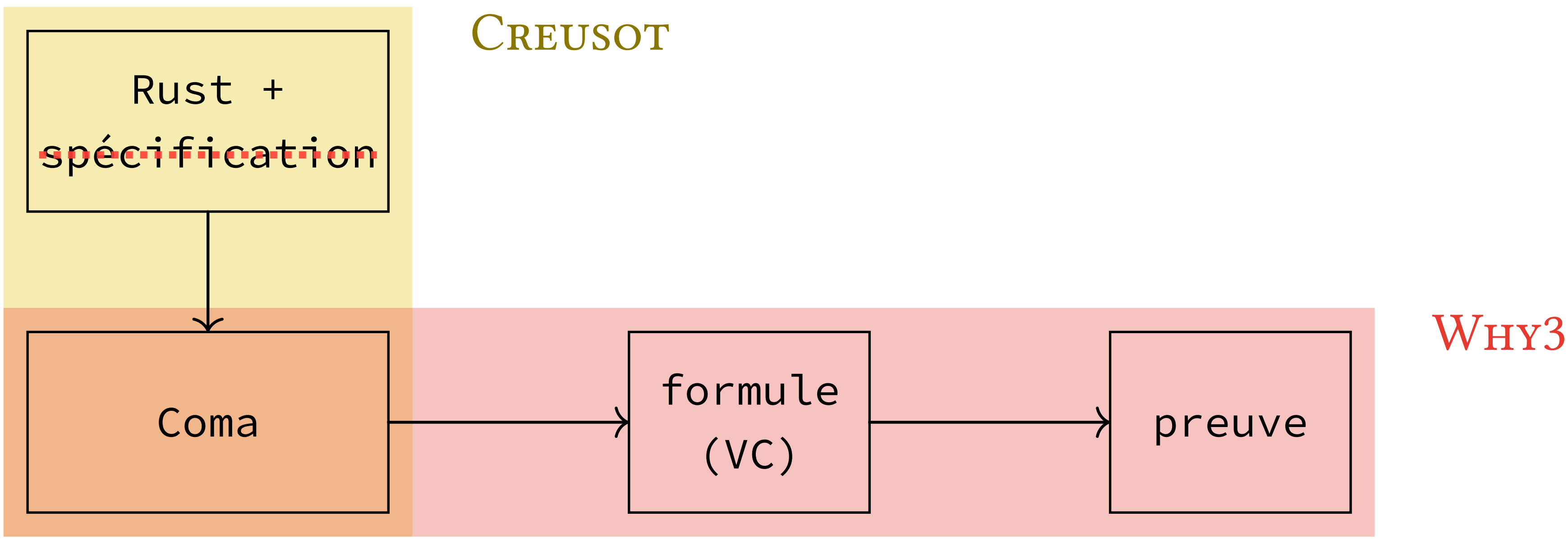
Creusot

Outil de vérification déductive pour Rust, basé sur Why3



Creusot

Outil de vérification déductive pour Rust, basé sur Why3



Creusot

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

Creusot

```
#[requires(∀x. opt = Some(x) ⟹ f.pre(x))]
#[ensures(∀x. opt = Some(x) ⟹
          ∃y. result = Some(y) ∧ 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(∀x. opt = Some(x) ⟹ f.pre(x))]
#[ensures(∀x. opt = Some(x) ⟹
          ∃y. result = Some(y) ∧ 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(∀x. opt = Some(x) ⟹
           i32::MIN ⩽ x * 2 ⩽ i32::MAX)]
#[ensures(∀x. opt = Some(x) ⟹
          ∃y. result = Some(y) ∧ y = x * 2)]
fn map_double(opt: Option<i32>) -> Option<i32> {
    …
}
```

On utilise directement la spécification donnée à `|x| 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
  $\hookrightarrow$ « 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	*mode appelant*
$\mathcal{D}(\uparrow e) \triangleq \mathcal{D}(e) \wedge \mathcal{A}(e)$	$\mathcal{A}(\uparrow e) \triangleq \top$
$\mathcal{D}(\{\varphi\}\, e) \triangleq \varphi \to \mathcal{D}(e)$	$\mathcal{A}(\{\varphi\}\, e) \triangleq \varphi \wedge (\varphi \to \mathcal{A}(e))$
$\mathcal{D}(h\ \overline{a}) \triangleq h^{\natural}\ \overline{a}$	$\mathcal{A}(h\ \overline{a}) \triangleq h\ \overline{a}$

VCgen de Coma (simplifié)

Générateur modal de conditions de vérification

mode définition

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

$$\mathcal{D}(\{\varphi\}\, e) \triangleq \varphi \to \mathcal{D}(e)$$

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

mode appelant

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

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

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

$$\widehat{\mathcal{A}}(\texttt{let}\ h\ \overline{a}\ \texttt{=}\ e) \triangleq \lambda \overline{a}.\ \mathcal{A}(e)$$

VCgen de Coma (simplifié)

Générateur modal de conditions de vérification

mode définition	*mode appelant*
$\mathcal{D}(\uparrow e) \triangleq \mathcal{D}(e) \land \mathcal{A}(e)$	$\mathcal{A}(\uparrow e) \triangleq \top$
$\mathcal{D}(\{\varphi\}\, e) \triangleq \varphi \to \mathcal{D}(e)$	$\mathcal{A}(\{\varphi\}\, e) \triangleq \varphi \land (\varphi \to \mathcal{A}(e))$
$\mathcal{D}(h\ \overline{a}) \triangleq h^{\natural}\ \overline{a}$	$\mathcal{A}(h\ \overline{a}) \triangleq h\ \overline{a}$

$$\widehat{\mathcal{A}}(\texttt{let}\ h\ \overline{a}\ \texttt{=}\ e) \triangleq \lambda\overline{a}.\ \mathcal{A}(e)$$

`double` … sans barrières ? oui !

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

pré- et postcondition

$$\hat{\mathcal{A}}(\texttt{let double } x \ ret \texttt{ = } e) = \lambda x : \texttt{i32}.\ \lambda ret : \texttt{i32} \to \text{Prop}.$$
$$(\texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max}) \land$$
$$ret\ (x \times 2)$$

pré- et postcondition

$$\hat{\mathcal{A}}(\texttt{let double } x \; ret \texttt{ = } e) = \lambda x : \texttt{i32}.\ \lambda ret : \texttt{i32} \to \mathrm{Prop}.$$

$$(\texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max}) \wedge$$

$$ret\ (x \times 2)$$

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

pré- et postcondition

$$\hat{\mathcal{A}}(\texttt{let double } x \ ret \texttt{ = } e) = \lambda x : \texttt{i32}.\ \lambda ret : \texttt{i32} \to \text{Prop}.$$
$$(\texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max}) \land$$
$$ret\ (x \times 2)$$

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

pré- et postcondition

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

$$(\texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max}) \wedge$$

$$ret\ (x \times 2)$$

WP(e, ⊤)

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

$$= \lambda x : \texttt{i32}.\ \texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max}$$

pré- et postcondition

$$\hat{\mathcal{A}}(\texttt{let double } x \; ret \texttt{ = } e) = \lambda x : \texttt{i32}.\ \lambda ret : \texttt{i32} \to \mathrm{Prop}.$$

$$(\texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max}) \land$$

$$ret\ (x \times 2)$$

$$\texttt{double'pre} = \overbrace{\lambda x : \texttt{i32}.\ \hat{\mathcal{A}}(\texttt{let double } x \; ret \texttt{ = } e)\ x\ (\texttt{fun \_} \mapsto \top)}^{\mathrm{WP}(e, \top)}$$

$$= \lambda x : \texttt{i32}.\ \texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max}$$

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

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

$\natural$: opérateur de « neutralisation »

pré- et postcondition

$$\natural\hat{\mathcal{A}}(\texttt{let double}\ x\ ret\ \texttt{=}\ e) = \lambda x : \texttt{i32}.\ \lambda ret : \texttt{i32} \to \text{Prop}.$$
$$\cancel{(\texttt{i32\_min} \le x \times 2 \le \texttt{i32\_max})} \land$$
$$ret\ (x \times 2)$$

$$\texttt{double'pre} = \overbrace{\lambda x : \texttt{i32}.\ \hat{\mathcal{A}}(\texttt{let double}\ x\ ret\ \texttt{=}\ e)\ x\ (\texttt{fun \_} \mapsto \top)}^{\text{WP(e, }\top\text{)}}$$
$$= \lambda x : \texttt{i32}.\ \texttt{i32\_min} \le x \times 2 \le \texttt{i32\_max}$$
$$\texttt{double'post} = \lambda x, r : \texttt{i32}.$$
$$\neg(\natural\hat{\mathcal{A}}(\texttt{let double}\ x\ ret\ \texttt{=}\ e)\ x\ (\texttt{fun y} \mapsto r \neq \texttt{y}))$$

$\natural$: opérateur de « neutralisation »

pré- et postcondition

$$\natural\hat{\mathcal{A}}(\texttt{let double } x \ ret \texttt{ = } e) = \lambda x : \texttt{i32}.\ \lambda ret : \texttt{i32} \to \text{Prop}.$$
$$\cancel{(\texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max})} \land$$
$$ret\ (x \times 2)$$

$$\texttt{double'pre} = \boxed{\lambda x : \texttt{i32}.\ \hat{\mathcal{A}}(\texttt{let double } x \ ret \texttt{ = } e)\ x\ (\texttt{fun \_} \mapsto \top)}^{\text{WP(e, }\top\text{)}}$$
$$= \lambda x : \texttt{i32}.\ \texttt{i32\_min} \leq x \times 2 \leq \texttt{i32\_max}$$
$$\texttt{double'post} = \lambda x, r : \texttt{i32}.$$
$$\neg(\natural\hat{\mathcal{A}}(\texttt{let double } x \ ret \texttt{ = } e)\ x\ (\texttt{fun y} \mapsto r \neq \texttt{y}))$$
$$= \lambda x, r : \texttt{i32}.\ r = x \times 2$$

$\natural$: opérateur de « neutralisation »

pré- et postcondition

$$\natural\hat{\mathcal{A}}(\texttt{let double } x \ ret \texttt{ = } e) = \lambda x : \texttt{i32}.\ \lambda ret : \texttt{i32} \to \mathrm{Prop}.$$
$$\cancel{(\texttt{i32\_min} \le x \times 2 \le \texttt{i32\_max})} \land$$
$$ret\ (x \times 2)$$

WP(e, ⊤)

$$\texttt{double'pre} = \boxed{\lambda x : \texttt{i32}.\ \hat{\mathcal{A}}(\texttt{let double } x \ ret \texttt{ = } e)\ x\ (\texttt{fun \_} \mapsto \top)}$$
$$= \lambda x : \texttt{i32}.\ \texttt{i32\_min} \le x \times 2 \le \texttt{i32\_max}$$
$$\texttt{double'post} = \boxed{\lambda x, r : \texttt{i32}.\ \neg(\natural\hat{\mathcal{A}}(\texttt{let double } x \ ret \texttt{ = } e)\ x\ (\texttt{fun y} \mapsto r \ne \texttt{y}))}$$
$$= \lambda x, r : \texttt{i32}.\ r = x \times 2$$

SP(e, WP(e, ⊤))

♮ : opérateur de « neutralisation »

Évaluation

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

```rust
#[requires(∀x. opt = Some(x) ⟹ f.pre(x))]
#[ensures(∀x. opt = Some(x) ⟹
          ∃y. result = Some(y) ∧ 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 ≤ i32::MAX)]
    #[requires(x * 2 ≥ i32::MIN)]
    #[ensures(result = x * 2)]
    |x| x * 2;
assert!(map(o, double).unwrap() == 42);
```

```
#[requires(∀x. opt = Some(x) ⟹ f.pre(x))]
#[ensures(∀x. opt = Some(x) ⟹
          ∃y. result = Some(y) ∧ 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

Bonus

VCgen de Coma (simplifié)

mode définition

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

$$\mathcal{D}(\{\varphi\}\, e) \triangleq \varphi \to \mathcal{D}(e)$$

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

$$\mathcal{D}(e\ /\ h\ \overline{a} = d) \triangleq$$
$$\mathtt{let}\ h\ \overline{a} = \mathcal{A}(d)\ \mathtt{in}\ \mathcal{D}(e)$$

mode appelant

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

$$\mathcal{A}(\{\varphi\}\, e) \triangleq \varphi^{\natural} \wedge (\varphi \to \mathcal{A}(e))$$

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

$$\mathcal{A}(e\ /\ h\ \overline{a} = d) \triangleq$$
$$\mathtt{let}\ h\ \overline{a} = \mathcal{A}(d)\ \mathtt{in}\ \mathcal{A}(e) \wedge \forall \overline{a}.\, \mathcal{D}(d)$$

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

$$\hat{\mathcal{D}}(\mathtt{let}\ h\ \overline{a} = e) \triangleq \forall \overline{a}.\ \mathcal{D}(e)$$

```
let x1 = opt1.map(
  #[requires(x@ + 1 <= i32::MAX@)]
  #[ensures(result@ == x@ + 1)]
  |x| x + 1,
);
let x2 = opt1.map(
  #[requires(2 * x@ >= i32::MIN@)]
  #[requires(2 * x@ <= i32::MAX@)]
  #[ensures(result.0@ == 2 * x@)]
  #[ensures(result.1  == x)]
  |x| (2 * x, x),
);
```

```
let x1 = opt1.map(|x| x + 1);

let x2 = opt1.map(|x| (2 * x, x));
```

```rust
let x = v1
    .iter()
    .map(
      #[requires(x@ < u32::MAX@)]
      #[ensures(result@ == x@ + 1)]
      |x| *x + 1,
    )
    .collect();

let y = v2
    .into_iter()
    .map(
      #[ensures(result == ! b)]
      |b| !b,
    )
    .collect();
```

```rust
let x = v1.iter().map(|x| *x + 1).collect();

let y = v2.into_iter().map(|b| !b).collect();
```