

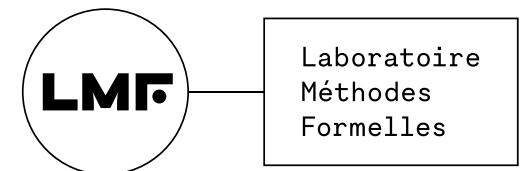
# Proving Rust closures

Paul Patault

*supervised by Andrei Paskevich and Jean-Christophe Filliâtre*

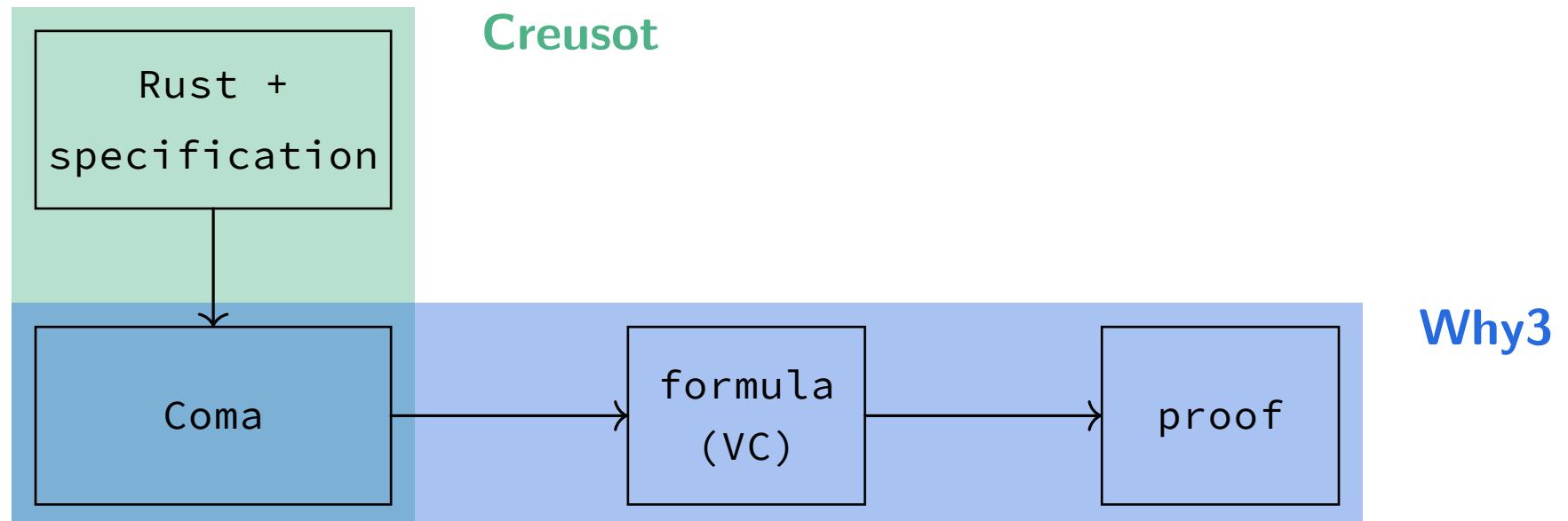
joint work with Arnaud Golfouse and Xavier Denis

June 2025 @ LMF goes green



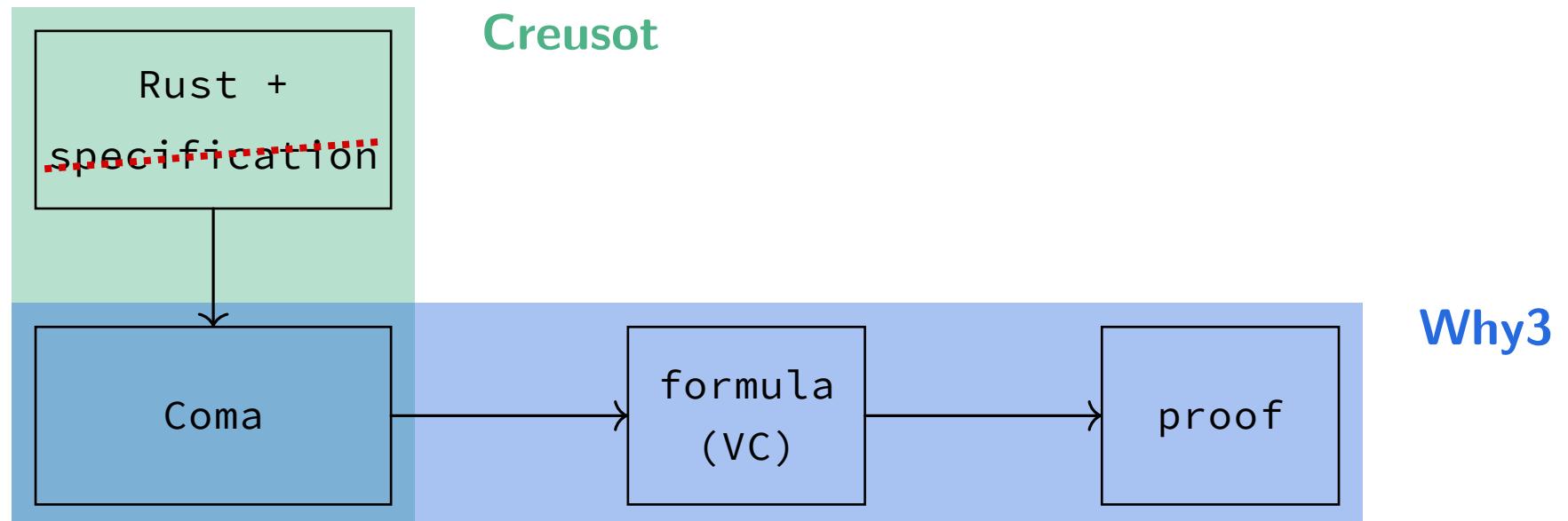
# Creusot *[Denis, Jourdan, Marché]*

Rust deductive verification tool, based upon Why3+Coma *[Filliâtre, Marché, Melquiond, Paskevich]*



# Creusot [Denis, Jourdan, Marché]

Rust deductive verification tool, based upon Why3+Coma [Filliâtre, Marché, Melquiond, Paskevich]



# Creusot [Denis, Jourdan, Marché]

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

# Creusot [Denis, Jourdan, Marché]

```
# [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 [Denis, Jourdan, Marché]

```
# [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);
```

# Problem

specification must be written

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

which is cumbersome...

# Problem

specification must be written

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

which is cumbersome... and easily incorrect

# Closure management

specification of `map` is monomorphized

```
# [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> {  
    ...  
}
```

⇒ only the specification of  $|x| \leq x * 2$  is used

# What we want

- do not write closures specifications
  - ↳ specification *is* the code
- keep track of predicates `f.pre` and `f.post` for monomorphization

# What we want

- do not write closures specifications
  - ↳ specification *is* the code
- keep track of predicates `f.pre` and `f.post` for monomorphization

Coma

# Coma

```
let double (x: i32) (k (r: i32)) =
  assert i32_min ≤ x * 2 ≤ i32_max ;
  hide out (x * 2)
where out (y: i32) =
  assert y = x * 2 ;
  hide k y
```

# Coma

```
let double (x: i32) (k (r: i32)) =
  assert i32_min ≤ x * 2 ≤ i32_max ;
  hide out (x * 2)
  where out (y: i32) =
    assert y = x * 2 ;
    hide k y
```

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

# pre- and postcondition

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

$$(\text{i32\_min} \leq x \times 2 \leq \text{i32\_max}) \wedge$$
$$k(\textcolor{blue}{x \times 2})$$

# pre- and postcondition

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

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

$k(\textcolor{blue}{x \times 2})$

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

# pre- and postcondition

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

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

$$k(\textcolor{blue}{x \times 2})$$

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

# pre- and postcondition

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

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

$$k(x \times 2)$$

WP(e,  $\top$ )  
↓

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

# pre- and postcondition

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

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

$$k(x \times 2)$$

WP(e, T)



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

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

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

$\natural$  : neutralization operation

# pre- and postcondition

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

$$(i32\_min \leq x \times 2 \leq i32\_max) \wedge \\ k(x \times 2)$$

$\boxed{\text{WP}(e, \top)}$

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

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

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

$\nexists$  : neutralization operation

# pre- and postcondition

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

$$\begin{array}{c} (\text{i32\_min} \leq x \times 2 \leq \text{i32\_max}) \wedge \\ k(x \times 2) \end{array} \quad \boxed{\text{WP}(e, \top)}$$

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

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

$\nexists$  : neutralization operation

# pre- and postcondition

$$\textcolor{red}{\natural} \mathcal{A}(\text{let double } x k = e) = \lambda x : \text{i32}. \lambda k : \text{i32} \rightarrow \text{Prop.}$$

$$\begin{array}{c} (\text{i32\_min} \leq x \times 2 \leq \text{i32\_max}) \wedge \\ k(x \times 2) \end{array} \quad \boxed{\text{WP}(e, \top)}$$

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

$$\begin{aligned} \text{double}'\text{post} &= \lambda x, r : \text{i32}. \\ &\quad \neg(\textcolor{red}{\natural} \mathcal{A}(\text{let double } x k = e) x (\text{fun } y \mapsto r \neq y)) \\ &= \lambda x, r : \text{i32}. \textcolor{blue}{r} = x \times 2 \end{aligned} \quad \boxed{\text{SP}(e, \text{WP}(e, \top))}$$

$\textcolor{red}{\natural}$  : neutralization operation

# Evaluation

example	no barrier	LoC	LoS	time (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

```

#[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( $\forall x. \text{opt} = \text{Some}(x) \Rightarrow f.\text{pre}(x)$ )]
#[ensures( $\forall x. \text{opt} = \text{Some}(x) \Rightarrow$ 
            $\exists y. \text{result} = \text{Some}(y) \wedge f.\text{post}(x, y)$ )]
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);

```

# What we have

- ✓ do not write closures specifications
- ✓ keep track of predicates f.pre and f.post for monomorphization

Thanks

# Bonus

```
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));
```

```
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();
```

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

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