Remonter les barrières pour ouvrir une clôture

Paul Patault   Arnaud Golfouse   Xavier Denis

lightning talk @ LMF

université PARIS-SACLAY
Inria
LMF Laboratoire Méthodes Formelles

Creusot

Outil de vérification déductive pour Rust

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

Creusot

Outil de vérification déductive pour 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)),
    }
}
```

Creusot

Outil de vérification déductive pour 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 = |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 fonction `map` est monorphisée, sa spécification aussi :

```
#[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 de `|x| x * 2`.

Ce qu'on veut

— ne pas écrire la spécification de la clôture

— conserver les prédicats `f.pre` et `f.post` pour monorphiser

Ce qu’on veut

— ne pas écrire la spécification de la clôture

— conserver les prédicats `f.pre` et `f.post` pour monorphiser

« Coma »

Short break...

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*
$\mathcal{D}(\uparrow e) \triangleq \mathcal{D}(e) \wedge \mathcal{A}(e)$	$\mathcal{A}(\uparrow e) \triangleq \top$
$\mathcal{D}(\{\varphi\}\, e) \triangleq \varphi \rightarrow \mathcal{D}(e)$	$\mathcal{A}(\{\varphi\}\, e) \triangleq \varphi \wedge (\varphi \rightarrow \mathcal{A}(e))$
$\mathcal{D}(h\ \overline{a}) \triangleq h^{\natural}\ \overline{a}$	$\mathcal{A}(h\ \overline{a}) \triangleq h\ \overline{a}$
$\widehat{\mathcal{D}}(\texttt{let}\ h\ \overline{a} = e) \triangleq \forall \overline{a}.\ \mathcal{D}(e)$	$\widehat{\mathcal{A}}(\texttt{let}\ h\ \overline{a} = e) \triangleq \lambda \overline{a}.\ \mathcal{A}(e)$

VCgen de Coma (simplifié)

Générateur modal de conditions de vérification

mode définition	*mode appelant*
$\mathcal{D}(\uparrow e) \triangleq \mathcal{D}(e) \wedge \mathcal{A}(e)$	$\mathcal{A}(\uparrow e) \triangleq \top$
$\mathcal{D}(\{\varphi\}\, e) \triangleq \varphi \to \mathcal{D}(e)$	$\mathcal{A}(\{\varphi\}\, e) \triangleq \varphi \wedge (\varphi \to \mathcal{A}(e))$
$\mathcal{D}(h\ \overline{a}) \triangleq h^{\natural}\ \overline{a}$	$\mathcal{A}(h\ \overline{a}) \triangleq h\ \overline{a}$
$\hat{\mathcal{D}}(\texttt{let}\ h\ \overline{a} = e) \triangleq \forall \overline{a}.\ \mathcal{D}(e)$	$\hat{\mathcal{A}}(\texttt{let}\ h\ \overline{a} = e) \triangleq \lambda \overline{a}.\ \mathcal{A}(e)$

`double` ... sans barrières ? oui !

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

pré- et postconditions

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

pré- et postconditions

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

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

pré- et postconditions

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

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

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

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

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

$$\texttt{double'post} = \lambda x, r : \texttt{i32}.\ \neg(\natural\hat{\mathcal{A}}(\texttt{let double ...})\ x\ (\texttt{fun y} \mapsto r \ne \texttt{y}))$$

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

$\natural$: opérateur de « neutralisation »

```
#[coma:extspec]
let double (x: i32) (ret (r: i32)) = ret (x * 2)

predicate double'pre (x: i32) =
  i32_min ⩽ x * 2 ⩽ i32_max

predicate double'post (x: i32) (r: i32) =
  r = x * 2

let map_double (o: option i32) (ret (r: option i32)) =
  assert {
    ∀x. o = Some(x) ⟹ double'pre x
  } ↑ unOpt o
        (-> out None)
        (x -> double x (y -> out (Some y)))
  / out (r: option i32) =
    assert {
      ∀x. o = Some(x) ⟹
      ∃y. result = Some(y) ∧ double'post x y
    } ↑ ret r
```

Ce qu'on a

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

Merci

Paul, Arnaud, Xavier