université
PARIS-SACLAY

UE14 · Vérification déductive de programmes

Programmation et spécification en WhyML

Master Informatique · FIIL · 2017 / 2018

1. Types de WhyML

Types de WhyML

WhyML supporte une grande partie de types d'OCaml :

- types polymorphes

```
type set 'a
```

- n-uplets :

```
type poly_pair 'a = ('a, 'a)
```

- enregistrements (*records*) :

```
type complex = { re : real; im : real }
```

- variants (*sum types*) :

```
type list 'a = Cons 'a (list 'a) | Nil
```

Types algébriques

Pour manipuler les types algébriques (enregistrements, variants) :

- accès aux champs des enregistrements :

```
let get_real (c : complex) = c.re
let use_imagination (c : complex) = im c
```

- reconstruction des enregistrements

```
let conjugate (c : complex) = { c with im = - c.im }
```

- analyse de motifs (*pattern matching*), sans `when` :

```
let rec length (l : list 'a) : int variant { l } =
  match l with
  | Cons _ ll -> 1 + length ll
  | Nil -> 0
  end
```

Les types abstraits doivent être axiomatisés :

```
theory Map
  type map 'a 'b

  function ([])   (a: map 'a 'b) (i: 'a): 'b
  function ([<-]) (a: map 'a 'b) (i: 'a) (v: 'b): map 'a 'b

  axiom Select_eq:
    forall m: map 'a 'b, k1 k2: 'a, v: 'b.
      k1 = k2  -> m[k1 <- v][k2] = v

  axiom Select_neq:
    forall m: map 'a 'b, k1 k2: 'a, v: 'b.
      k1 <> k2 -> m[k1 <- v][k2] = m[k2]
end
```

Types abstraits (cont.)

Les types abstraits doivent être axiomatisés :

```
theory Set
  type set 'a
  predicate mem 'a (set 'a)

  predicate (==) (s1 s2: set 'a) =
    forall x: 'a. mem x s1 <-> mem x s2
  axiom extensionality:
    forall s1 s2: set 'a. s1 == s2 -> s1 = s2

  predicate subset (s1 s2: set 'a) =
    forall x: 'a. mem x s1 -> mem x s2
  lemma subset_refl: forall s: set 'a. subset s s

  constant empty : set 'a
  axiom empty_def: forall x: 'a. not (mem x empty)
  ...
```

Langage logique de WHYML

- les mêmes types sont disponibles dans la logique
- `match-with-end`, `if-then-else`, `let-in`
  sont acceptés dans les termes et les formules
- fonctions et prédicats peuvent être définis récursivement :

```
predicate mem (x: 'a) (l: list 'a) = match l with
  Cons y r -> x = y \/ mem x r | Nil -> false end
```

  pas de `variant`s, WHY3 exige la décroissance structurelle

- prédicats inductifs (utile pour définir les clôtures transitives) :

```
inductive sorted (l: list int) =
  | SortedNil: sorted Nil
  | SortedOne: forall x: int. sorted (Cons x Nil)
  | SortedTwo: forall x y: int, l: list int.
      x <= y -> sorted (Cons y l) ->
        sorted (Cons x (Cons y l))
```

2. Code fantôme (*ghost code*)

Code fantôme : exemple

Calculer les nombres de Fibonacci avec une fonction récursive en $O(n)$:

```
let rec aux (a b n: int): int
  requires { 0 <= n }
  requires {                                                    }
  ensures  {

                                                                }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n

(* fib_rec 5 = aux 0 1 5 = aux 1 1 4 = aux 1 2 3 =
               aux 2 3 2 = aux 3 5 1 = aux 5 8 0 = 5 *)
```

Code fantôme : exemple

Calculer les nombres de Fibonacci avec une fonction récursive en $O(n)$:

```
let rec aux (a b n: int): int
  requires { 0 <= n }
  requires { exists k. 0 <= k /\ a = fib k /\ b = fib (k+1) }
  ensures  { exists k. 0 <= k /\ a = fib k /\ b = fib (k+1) /\
                                          result = fib (k+n) }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n

(* fib_rec 5 = aux 0 1 5 = aux 1 1 4 = aux 1 2 3 =
               aux 2 3 2 = aux 3 5 1 = aux 5 8 0 = 5 *)
```

Code fantôme : exemple

À la place des existentiels on peut utiliser un paramètre fantôme :

```
let rec aux (a b n: int) (ghost k: int): int
  requires { 0 <= n }
  requires { 0 <= k /\ a = fib k /\ b = fib (k+1) }
  ensures  { result = fib (k+n) }
  variant  { n }
= if n = 0 then a else aux b (a+b) (n-1) (k+1)

let fib_rec (n: int): int
  requires { 0 <= n }
  ensures  { result = fib n }
= aux 0 1 n 0
```

L'esprit du code fantôme

Le code fantôme sert à la spécification et à la preuve

$\Rightarrow$ le principe de non interférence :

On doit pouvoir éliminer le code fantôme
sans modifier le résultat du programme.

L'esprit du code fantôme

Le code fantôme sert à la spécification et à la preuve

$\Rightarrow$ le principe de non interférence :

> On doit pouvoir éliminer le code fantôme
> sans modifier le résultat du programme.

Par conséquence :

- le code normal ne lit pas des données fantômes
  - si k est fantôme, alors $(k+1)$ l'est aussi

L'esprit du code fantôme

Le code fantôme sert à la spécification et à la preuve

$\Rightarrow$ le principe de non interférence :

> On doit pouvoir éliminer le code fantôme
> sans modifier le résultat du programme.

Par conséquence :

- le code normal ne lit pas des données fantômes
  - si k est fantôme, alors $(k+1)$ l'est aussi
- le code fantôme ne modifie pas des données normales
  - si r est une référence normale, alors $r := \texttt{ghost}\ k$ est interdit

L'esprit du code fantôme

Le code fantôme sert à la spécification et à la preuve

$\Rightarrow$ le principe de non interférence :

> On doit pouvoir éliminer le code fantôme
> sans modifier le résultat du programme.

Par conséquence :

- le code normal ne lit pas des données fantômes
  - si k est fantôme, alors $(k+1)$ l'est aussi
- le code fantôme ne modifie pas des données normales
  - si r est une référence normale, alors $r := \texttt{ghost}\ k$ est interdit
- le code fantôme ne change pas le flot de contrôle du code normale
  - si c est fantôme, `if` c `then ...` et `while` c `do ...` le sont aussi

L'esprit du code fantôme

Le code fantôme sert à la spécification et à la preuve

$\Rightarrow$ le principe de non interférence :

On doit pouvoir éliminer le code fantôme
sans modifier le résultat du programme.

Par conséquence :

- le code normal ne lit pas des données fantômes
  - si k est fantôme, alors $(k+1)$ l'est aussi
- le code fantôme ne modifie pas des données normales
  - si r est une référence normale, alors $r := \texttt{ghost}\ k$ est interdit
- le code fantôme ne change pas le flot de contrôle du code normale
  - si c est fantôme, `if` c `then ...` et `while` c `do ...` le sont aussi
- le code fantôme termine
  - `while true do skip done ; assert false` est prouvable

Code fantôme dans WhyML

Peuvent être déclarés fantômes :

- paramètres des fonctions

```
val aux (a b n: int) (ghost k: int): int
```

Code fantôme dans WHYML

Peuvent être déclarés fantômes :

- paramètres des fonctions

```
val aux (a b n: int) (ghost k: int): int
```

- champs des enregistrements et des variants

```
type queue 'a = { head: list 'a;  (* get from head *)
                  tail: list 'a;  (* add to tail *)
            ghost elts: list 'a;  (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

Code fantôme dans WHYML

Peuvent être déclarés fantômes :

- paramètres des fonctions

```
val aux (a b n: int) (ghost k: int): int
```

- champs des enregistrements et des variants

```
type queue 'a = { head: list 'a;  (* get from head *)
                  tail: list 'a;  (* add to tail *)
            ghost elts: list 'a;  (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

- variables et fonctions

```
let ghost x = qu.elts in ...
let ghost rev_elts qu = qu.tail ++ reverse qu.head
```

Code fantôme dans WHYML

Peuvent être déclarés fantômes :

- paramètres des fonctions

```
val aux (a b n: int) (ghost k: int): int
```

- champs des enregistrements et des variants

```
type queue 'a = { head: list 'a;  (* get from head *)
                  tail: list 'a;  (* add to tail *)
            ghost elts: list 'a;  (* logical view *) }
invariant { self.elts = self.head ++ reverse self.tail }
```

- variables et fonctions

```
let ghost x = qu.elts in ...
let ghost rev_elts qu = qu.tail ++ reverse qu.head
```

- expressions de programme

```
let x = ghost qu.elts in ...
```

Comment ça marche ?

Le code fantôme et le code visible sont faits des mêmes éléments.

Annoter explicitement toute expression fantôme serait bien lourd.

Comment ça marche ?

Le code fantôme et le code visible sont faits des mêmes éléments.

Annoter explicitement toute expression fantôme serait bien lourd.

Solution : Modifier le système de types et utiliser l'inférence.

$$\Gamma \vdash e : \varsigma$$

ς — `int`, `bool`, `unit` (listes, tableaux, etc.)

Comment ça marche ?

Le code fantôme et le code visible sont faits des mêmes éléments.

Annoter explicitement toute expression fantôme serait bien lourd.

Solution : Modifier le système de types et utiliser l'inférence.

$$\Gamma \vdash e : \varsigma \cdot \varepsilon$$

- ς — `int`, `bool`, `unit` (listes, tableaux, etc.)
- ε — effets de bord potentiels

références	r `:= ...`, `let` $r =$ `ref ... in`
exceptions levées	`raise` E, `try ... with` E $\rightarrow$
divergence	terminaison non prouvée

Comment ça marche ?

Le code fantôme et le code visible sont faits des mêmes éléments.

Annoter explicitement toute expression fantôme serait bien lourd.

Solution : Modifier le système de types et utiliser l'inférence.

$$\Gamma \vdash e : \varsigma \cdot \varepsilon \cdot \mathfrak{g}$$

- ς — `int`, `bool`, `unit` (listes, tableaux, etc.)
- ε — effets de bord potentiels

références	`r := ...`, `let r = ref ... in`
exceptions levées	`raise` E, `try ... with` E $\rightarrow$
divergence	terminaison non prouvée

- $\mathfrak{g}$ — expression visible ou fantôme ?

Comment ça marche ?

Le code fantôme et le code visible sont faits des mêmes éléments.

Annoter explicitement toute expression fantôme serait bien lourd.

Solution : Modifier le système de types et utiliser l'inférence.

$$\Gamma \vdash e : \varsigma \cdot \varepsilon \cdot \mathfrak{g} \cdot \mathfrak{m}$$

- ς — `int`, `bool`, `unit` (listes, tableaux, etc.)
- ε — effets de bord potentiels
  - références — *r* `:=` `...`, `let` *r* = `ref` `...` `in`
  - exceptions levées — `raise` E, `try` `...` `with` E $\rightarrow$
  - divergence — terminaison non prouvée
- $\mathfrak{g}$ — expression visible ou fantôme ?
- $\mathfrak{m}$ — résultat visible ou fantôme ?

Qui est fantôme ?

Toute variable ou référence est considérée fantôme

- si déclarée fantôme explicitement : `let ghost v^g = 6 * 6 in ...`
- si initialisée avec une valeur fantôme : `let r^g = ref v^g in ...`
- si déclarée dans un bloc `ghost` : `ghost ( let x^g = 42 in ... )`

Qui est fantôme ?

Toute variable ou référence est considérée fantôme

- si déclarée fantôme explicitement : `let ghost v^g = 6 * 6 in ...`
- si initialisée avec une valeur fantôme : `let r^g = ref v^g in ...`
- si déclarée dans un bloc `ghost` : `ghost ( let x^g = 42 in ... )`

1. terme t est fantôme $\equiv$ t contient une variable ou référence fantôme

Qui est fantôme ?

Toute variable ou référence est considérée fantôme

- si déclarée fantôme explicitement : `let ghost v^g = 6 * 6 in ...`
- si initialisée avec une valeur fantôme : `let r^g = ref v^g in ...`
- si déclarée dans un bloc `ghost` : `ghost ( let x^g = 42 in ... )`

1. terme t est fantôme $\equiv$ t contient une variable ou référence fantôme
2. r `:=` t est fantôme $\equiv$ r est une référence fantôme (Q : pas t ?)

Qui est fantôme ?

Toute variable ou référence est considérée fantôme

- si déclarée fantôme explicitement : `let ghost` $\mathtt{v}^{\mathtt{g}}$ `= 6 * 6 in ...`
- si initialisée avec une valeur fantôme : `let` $\mathtt{r}^{\mathtt{g}}$ `= ref` $\mathtt{v}^{\mathtt{g}}$ `in ...`
- si déclarée dans un bloc `ghost` : `ghost (let` $\mathtt{x}^{\mathtt{g}}$ `= 42 in ... )`

1. terme t est fantôme $\equiv$ t contient une variable ou référence fantôme
2. r `:=` t est fantôme $\equiv$ r est une référence fantôme (Q : pas t ?)
3. `skip` n'est pas fantôme

Qui est fantôme ?

Toute variable ou référence est considérée fantôme

- si déclarée fantôme explicitement : `let ghost v^g = 6 * 6 in ...`
- si initialisée avec une valeur fantôme : `let r^g = ref v^g in ...`
- si déclarée dans un bloc `ghost` : `ghost ( let x^g = 42 in ... )`

1. terme t est fantôme $\equiv$ t contient une variable ou référence fantôme
2. r `:=` t est fantôme $\equiv$ r est une référence fantôme (Q : pas t ?)
3. `skip` n'est pas fantôme
4. `raise E` n'est pas fantôme

Qui est fantôme ?

Toute variable ou référence est considérée fantôme

- si déclarée fantôme explicitement : `let ghost v^g = 6 * 6 in ...`
- si initialisée avec une valeur fantôme : `let r^g = ref v^g in ...`
- si déclarée dans un bloc `ghost` : `ghost ( let x^g = 42 in ... )`

1. terme t est fantôme $\equiv$ t contient une variable ou référence fantôme
2. r `:=` t est fantôme $\equiv$ r est une référence fantôme (Q : pas t ?)
3. `skip` n'est pas fantôme
4. `raise E` n'est pas fantôme

   sauf si on passe une valeur fantôme avec `E` : `raise E v^g`

Qui est fantôme ?

Toute variable ou référence est considérée fantôme

- si déclarée fantôme explicitement : `let ghost v^g = 6 * 6 in ...`
- si initialisée avec une valeur fantôme : `let r^g = ref v^g in ...`
- si déclarée dans un bloc `ghost` : `ghost ( let x^g = 42 in ... )`

1. terme t est fantôme $\equiv$ t contient une variable ou référence fantôme
2. r `:=` t est fantôme $\equiv$ r est une référence fantôme (Q : pas t ?)
3. `skip` n'est pas fantôme
4. `raise E` n'est pas fantôme

   sauf si on passe une valeur fantôme avec `E` : `raise E v^g`

   sauf si `E` transporte des valeurs fantômes : `exception E (ghost int)`

Qui est fantôme ?

Une expression e a un effet visible si et seulement si

- e modifie une référence visible
- e diverge ($=$ la terminaison n'est pas vérifiée)
- e n'est pas fantôme et levé une exception

Qui est fantôme ?

Une expression e a un effet visible si et seulement si

- e modifie une référence visible
- e diverge (= la terminaison n'est pas vérifiée)
- e n'est pas fantôme et levé une exception

5. e_1 `;` e_2 / `let` $v = e_1$ `in` e_2 / `let` $v =$ `ref` e_1 `in` e_2 est fantôme $\equiv$
   - e_2 est fantôme et e_1 n'a pas d'effets visibles (Q : et sinon ?)
   - e_1 ou e_2 est fantôme et lève une exception (Q : pourquoi ?)

Qui est fantôme ?

Une expression e a un effet visible si et seulement si

- e modifie une référence visible
- e diverge (= la terminaison n'est pas vérifiée)
- e n'est pas fantôme et levé une exception

5. e_1 `;` e_2 / `let` $v = e_1$ `in` e_2 / `let` $v =$ `ref` e_1 `in` e_2 est fantôme $\equiv$
   - e_2 est fantôme et e_1 n'a pas d'effets visibles (Q : et sinon ?)
   - e_1 ou e_2 est fantôme et lève une exception (Q : pourquoi ?)
6. `try` e_1 `with` E $\rightarrow$ e_2 / `try` e_1 `with` E $v \rightarrow$ e_2 est fantôme $\equiv$
   - e_1 est fantôme
   - e_2 est fantôme et lève une exception

Qui est fantôme ?

Une expression e a un effet visible si et seulement si

- e modifie une référence visible
- e diverge (= la terminaison n'est pas vérifiée)
- e n'est pas fantôme et levé une exception

7. `if` t `then` e_1 `else` e_2 est fantôme $\equiv$
   - t est fantôme (sauf si e_1 ou e_2 est `assert false`)
   - e_1 est fantôme et e_2 n'a pas d'effets visibles
   - e_2 est fantôme et e_1 n'a pas d'effets visibles
   - e_1 ou e_2 est fantôme et levé une exception

Qui est fantôme ?

Une expression e a un effet visible si et seulement si

- e modifie une référence visible
- e diverge (= la terminaison n'est pas vérifiée)
- e n'est pas fantôme et levé une exception

7. `if` t `then` e_1 `else` e_2 est fantôme $\equiv$
   - t est fantôme (sauf si e_1 ou e_2 est `assert false`)
   - e_1 est fantôme et e_2 n'a pas d'effets visibles
   - e_2 est fantôme et e_1 n'a pas d'effets visibles
   - e_1 ou e_2 est fantôme et levé une exception
8. `while` t `do` e `done` est fantôme $\equiv$ t ou e est fantôme

Qui est fantôme ?

9. appel de fonction $f\ t_1\ \ldots\ t_n$ est fantôme $\equiv$
   - function f est fantôme ou un argument t_i est fantôme
     sauf si f attend un paramètre fantôme dans cette position

Qui est fantôme ?

9. appel de fonction $f\ t_1\ \ldots\ t_n$ est fantôme $\equiv$
   - function f est fantôme ou un argument t_i est fantôme
     sauf si f attend un paramètre fantôme dans cette position

Dans une définition de fonction

- tout paramètre fantôme est déclaré explicitement
- nous pouvons inférer le statut fantôme de toute sous-expression

Qui est fantôme ?

9. appel de fonction $f\ t_1\ \ldots\ t_n$ est fantôme $\equiv$
   - function f est fantôme ou un argument t_i est fantôme
     sauf si f attend un paramètre fantôme dans cette position

Dans une définition de fonction

- tout paramètre fantôme est déclaré explicitement
- nous pouvons inférer le statut fantôme de toute sous-expression

$\lceil\cdot\rceil$ convertit les données fantômes en `unit` et le code fantôme en `skip`.

Theorem* : L'opérateur $\lceil\cdot\rceil$ préserve la sémantique visible du programme.

$$\begin{array}{ccc} e\cdot\mu & \longrightarrow^{\star} & c\cdot\mu' \\ \Downarrow & & \Downarrow \\ \lceil e\rceil\cdot\lceil\mu\rceil & \longrightarrow^{\star} & \lceil c\rceil\cdot\lceil\mu'\rceil \end{array} \qquad \begin{array}{ccc} e\cdot\mu & \Longrightarrow & \infty \\ \Downarrow & & \Downarrow \\ \lceil e\rceil\cdot\lceil\mu\rceil & \Longrightarrow & \infty \end{array}$$

Fonctions-lemmes

Idée générale : une fonction $f\ \vec{x}$ `requires` P_f `ensures` Q_f qui

- ne renvoie pas de résultats
- ne produit pas d'effet de bord
- termine

fournit une preuve constructive de $\forall \vec{x}. P_f \longrightarrow Q_f$

$\Rightarrow$ une fonction pure récursive simule une preuve par induction

Fonctions-lemmes

Idée générale : une fonction $f\ \vec{x}$ `requires` P_f `ensures` Q_f qui

- ne renvoie pas de résultats
- ne produit pas d'effet de bord
- termine

fournit une preuve constructive de $\forall \vec{x}. P_f \rightarrow Q_f$

$\Rightarrow$ une fonction pure récursive simule une preuve par induction

```
function rev_append (l r: list 'a): list 'a = match l with
  | Cons a ll -> rev_append ll (Cons a r)  |  Nil -> r end

let rec lemma length_rev_append (l r: list 'a)  variant {l}
  ensures { length (rev_append l r) = length l + length r }
=
  match l with Cons a ll -> length_rev_append ll (Cons a r)
               | Nil -> () end
```

Fonctions-lemmes

```
function rev_append (l r: list 'a): list 'a = match l with
  | Cons a ll -> rev_append ll (Cons a r)  |  Nil -> r end

let rec lemma length_rev_append (l r: list 'a)  variant {l}
  ensures { length (rev_append l r) = length l + length r }
=
  match l with Cons a ll -> length_rev_append ll (Cons a r)
               | Nil -> () end
```

- par la postcondition de l'appel récursif :

  ```
  length (rev_append ll (Cons a r)) = length ll + length (Cons a r)
  ```

- par définition de `rev_append` :

  ```
  rev_append (Cons a ll) r = rev_append ll (Cons a r)
  ```

- par définition de `length` :

  ```
  length (Cons a ll) + length r = length ll + length (Cons a r)
  ```

3. Données mutables

Enregistrements avec des champs mutables

```
module Ref
  type ref 'a = { mutable contents : 'a } (* comme en OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

Enregistrements avec des champs mutables

```
module Ref
  type ref 'a = { mutable contents : 'a } (* comme en OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- peuvent être passés en argument et renvoyés en valeur

Enregistrements avec des champs mutables

```
module Ref
  type ref 'a = { mutable contents : 'a } (* comme en OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- peuvent être passés en argument et renvoyés en valeur
- peuvent être créés localement et déclarés globalement
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`

Enregistrements avec des champs mutables

```
module Ref
  type ref 'a = { mutable contents : 'a } (* comme en OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- peuvent être passés en argument et renvoyés en valeur
- peuvent être créés localement et déclarés globalement
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- peuvent stocker des données fantômes
  - `let ghost r := ref 42 in ... ghost (r := -r) ...`

Enregistrements avec des champs mutables

```
module Ref
  type ref 'a = { mutable contents : 'a } (* comme en OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- peuvent être passés en argument et renvoyés en valeur
- peuvent être créés localement et déclarés globalement
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- peuvent stocker des données fantômes
  - `let ghost r := ref 42 in ... ghost (r := -r) ...`
- interdits dans les structures récursifs : pas de `list (ref 'a)`

Enregistrements avec des champs mutables

```
module Ref
  type ref 'a = { mutable contents : 'a } (* comme en OCaml *)
  function (!) (r: ref 'a) : 'a = r.contents
  let ref (v: 'a) = { contents = v }
  let (!) (r:ref 'a) = r.contents
  let (:=) (r:ref 'a) (v:'a) = r.contents <- v
end
```

- peuvent être passés en argument et renvoyés en valeur
- peuvent être créés localement et déclarés globalement
  - `let r = ref 0 in while !r < 42 do r := !r + 1 done`
  - `val gr : ref int`
- peuvent stocker des données fantômes
  - `let ghost r := ref 42 in ... ghost (r := -r) ...`
- interdits dans les structures récursifs : pas de `list (ref 'a)`
- interdits sous les types abstraits : pas de `set (ref 'a)`

Problème d'alias

```
let double_incr (s1 s2: ref int): unit writes {s1,s2}
  ensures { !s1 = 1 + old !s1 /\ !s2 = 2 + old !s2 }
= s1 := 1 + !s1; s2 := 2 + !s2

let wrong () =
  let s = ref 0 in
  double_incr s s;   (* alias écriture/écriture *)
  assert { !s = 1 /\ !s = 2 }   (* en fait, !s = 3 *)
```

Problème d'alias

```
let double_incr (s1 s2: ref int): unit writes {s1,s2}
  ensures { !s1 = 1 + old !s1 /\ !s2 = 2 + old !s2 }
= s1 := 1 + !s1; s2 := 2 + !s2

let wrong () =
  let s = ref 0 in
  double_incr s s;   (* alias écriture/écriture *)
  assert { !s = 1 /\ !s = 2 }   (* en fait, !s = 3 *)
```

```
val g : ref int

let set_from_g (r: ref int): unit writes {r}
  ensures { !r = !g + 1 }
= r := !g + 1

let wrong () =
  set_from_g g;   (* alias lecture/écriture *)
  assert { !g = !g + 1 }   (* contradiction *)
```

WP vs. alias

La règle de WP standard pour l'affectation :

$$\mathrm{WP}(x := 42, Q[x, y, z]) = Q[42, y, z]$$

Mais si x et z sont deux noms pour la même référence

$$\mathrm{WP}(x := 42, Q[x, y, z]) \quad \text{doit être} \quad Q[42, y, 42]$$

Problème : Connaître *statiquement* quand deux valeurs sont aliasées.

WP vs. alias

La règle de WP standard pour l'affectation :

$$\mathrm{WP}(x := 42, Q[x, y, z]) = Q[42, y, z]$$

Mais si x et z sont deux noms pour la même référence

$$\mathrm{WP}(x := 42, Q[x, y, z]) \quad \text{doit être} \quad Q[42, y, 42]$$

Problème : Connaître *statiquement* quand deux valeurs sont aliasées.

Solution : Modifier le système de types et utiliser l'inférence.

WP avec des alias

Tout type mutable porte un *token d'identité invisible* — une région :

$$x : \texttt{ref}\ \rho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \rho\ \texttt{int}$$

WP avec des alias

Tout type mutable porte un *token d'identité invisible* — une région :

$$x : \texttt{ref}\ \rho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \rho\ \texttt{int}$$

Certains programmes ne sont plus typables : $\texttt{if}\ \ldots\ \texttt{then}\ x\ \texttt{else}\ y$: ?

WP avec des alias

Tout type mutable porte un *token d'identité invisible* — une région :

$$x : \texttt{ref}\ \rho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \rho\ \texttt{int}$$

Certains programmes ne sont plus typables : $\texttt{if}\ \ldots\ \texttt{then}\ x\ \texttt{else}\ y$: ?

et tant mieux : $\mathrm{WP}(\texttt{let}\ r = x\ \textit{ou peut-être}\ y\ \texttt{in}\ r := 42, Q[x, y, z]) = ?$

WP avec des alias

Tout type mutable porte un *token d'identité invisible* — une région :

$$x : \texttt{ref}\ \rho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \rho\ \texttt{int}$$

Certains programmes ne sont plus typables : `if ... then` x `else` y : ?

et tant mieux : $\mathrm{WP}(\texttt{let}\ r = x\ \textit{ou peut-être}\ y\ \texttt{in}\ r := 42, Q[x, y, z]) = ?$

L'inférence de types à la ML révèle l'identité de toute sous-expression

- les paramètres formels et les références globales sont forcement séparés

WP avec des alias

Tout type mutable porte un *token d'identité invisible* — une région :

$$x : \texttt{ref}\ \rho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \rho\ \texttt{int}$$

Certains programmes ne sont plus typables : `if ... then` x `else` y : ?

et tant mieux : $\mathrm{WP}(\texttt{let}\ r = x\ \textit{ou peut-être}\ y\ \texttt{in}\ r := 42, Q[x, y, z]) = ?$

L'inférence de types à la ML révèle l'identité de toute sous-expression

- les paramètres formels et les références globales sont forcement séparés

La règle de WP pour l'affection, révisée : $\mathrm{WP}(x_\tau := t, Q) = Q\sigma$

où σ remplace dans Q toute variable $y : \pi[\tau]$ avec la valeur « à jour »

- un alias de x peut être stocké dans une référence ou un enregistrement

Peut-on aller plus loin ?

Un tableau rédimensionnable du pauvre :

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ρ (array ρ1 int) *)
```

Peut-on aller plus loin ?

Un tableau rédimensionnable du pauvre :

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ρ (array ρ1 int) *)
```

Changeons sa taille :

```
let olda = !resa (* olda : array ρ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa := newa (* newa : array ρ2 int *)
```

Peut-on aller plus loin ?

Un tableau rédimensionnable du pauvre :

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ρ (array ρ1 int) *)
```

Changeons sa taille :

```
let olda = !resa (* olda : array ρ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa := newa (* newa : array ρ2 int *)
```

Erreur de typage : Nous cassons la correspondance regions $\leftrightarrow$ aliases !

Peut-on aller plus loin ?

Un tableau rédimensionnable du pauvre :

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ρ (array ρ1 int) *)
```

Changeons sa taille :

```
let olda = !resa (* olda : array ρ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa := newa (* newa : array ρ2 int *)
```

Erreur de typage : Nous cassons la correspondance regions $\leftrightarrow$ aliases !

Changer le type de `resa` ? Que faire pour `if ... then resa := newa` ?

Oui, on peut !

Laissons chacun garder son type :

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ρ (array ρ1 int) *)
let olda = !resa (* olda : array ρ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa.contents ← newa (* newa : array ρ2 int *)
```

`newa`, `olda` — les témoins de la corruption du système de types

Oui, on peut !

Laissons chacun garder son type :

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ρ (array ρ1 int) *)
let olda = !resa (* olda : array ρ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa.contents ← newa (* newa : array ρ2 int *)
```

`newa`, `olda` — les témoins de la corruption du système de types

Que fait-on avec des témoins indésirables ? — A.G. CAPONE

Oui, on peut !

Laissons chacun garder son type :

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ρ (array ρ1 int) *)
let olda = !resa (* olda : array ρ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa.contents ← newa (* newa : array ρ2 int *)
```

Les expressions qui changent les types obtiennent un nouveau effet :

$$\texttt{writes}\ \rho \cdot \texttt{resets}\ \rho_1, \rho_2$$

e_1 ; e_2 est bien typée $\Rightarrow$ dans toute variable libre de e_2,
toute région réinitialisée par e_1 intervient sous une région modifiée par e_1

Oui, on peut !

Laissons chacun garder son type :

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ρ (array ρ1 int) *)
let olda = !resa (* olda : array ρ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa.contents ← newa (* newa : array ρ2 int *)
```

Les expressions qui changent les types obtiennent un nouveau effet :

$$\texttt{writes}\ \rho \cdot \texttt{resets}\ \rho_1, \rho_2$$

e_1 ; e_2 est bien typée $\Rightarrow$ dans toute variable libre de e_2,
toute région réinitialisée par e_1 intervient sous une région modifiée par e_1

Ainsi : `resa` et ses alias peuvent survivre, mais `olda` et `newa` sont tués.

L'effet tueur

e_1 ; e_2 est bien typée $\Rightarrow$ dans toute variable libre de e_2,
toute région réinitialisée par e_1 intervient sous une région modifiée par e_1

L'effet tueur

e_1 ; e_2 est bien typée $\Rightarrow$ dans toute variable libre de e_2,
toute région réinitialisée par e_1 intervient sous une région modifiée par e_1

L'effet de reset exprime aussi la fraîcheur :

> Si on crée une valeur mutable fraîche et lui affecte la région ρ,
> on doit tuer toute variables existante dont le type contient ρ.

L'effet tueur

e_1 ; e_2 est bien typée $\Rightarrow$ dans toute variable libre de e_2,
toute région réinitialisée par e_1 intervient sous une région modifiée par e_1

L'effet de reset exprime aussi la fraîcheur :

> Si on crée une valeur mutable fraîche et lui affecte la région ρ,
> on doit tuer toute variables existante dont le type contient ρ.

Union d'effets (pour séquence ou branchement) :

> x_τ survie $\varepsilon_1 \sqcup \varepsilon_2$ $\Leftrightarrow$ x_τ survie ε_1 et ε_2.

L'effet tueur

e_1 ; e_2 est bien typée $\Rightarrow$ dans toute variable libre de e_2,
toute région réinitialisée par e_1 intervient sous une région modifiée par e_1

L'effet de reset exprime aussi la fraîcheur :

> Si on crée une valeur mutable fraîche et lui affecte la région ρ,
> on doit tuer toute variables existante dont le type contient ρ.

Union d'effets (pour séquence ou branchement) :

x_τ survie $\varepsilon_1 \sqcup \varepsilon_2$ $\Leftrightarrow$ x_τ survie ε_1 et ε_2.

Ainsi :

- les régions réinitialisées de ε_1 et ε_2 s'ajoutent,
- les régions modifiées de ε_i tuées par ε_{2-i} sont ignorées.

Résumé

Le calcul de WP standard interdit les alias.

- au moins pour les valeurs modifiées
- WHY3 relâche cette restriction grâce au contrôle statique d'alias

Résumé

Le calcul de WP standard interdit les alias.

- au moins pour les valeurs modifiées
- WHY3 relâche cette restriction grâce au contrôle statique d'alias

Le programmeur doit indiquer les dépendances externes des fonctions :

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- sinon le contrôle statique d'alias n'a pas assez d'information

Résumé

Le calcul de WP standard interdit les alias.

- au moins pour les valeurs modifiées
- WHY3 relâche cette restriction grâce au contrôle statique d'alias

Le programmeur doit indiquer les dépendances externes des fonctions :

- `val set_from_g (r: ref int): unit writes {r} reads {g}`
- sinon le contrôle statique d'alias n'a pas assez d'information

Pour vérifier des programmes avec des pointeurs arbitraires
on a besoin d'approches plus sophistiquées

- modèle de mémoire (par exemple, tableaux « adresse $\longrightarrow$ valeur »)
- gérer les alias dans les VC : logique de séparation, *dynamic frames*...
- voir la 2^{e} partie du cours

Spécification abstraites

Restrictions sur les alias dans WHYML

$\Rightarrow$ certaines structures ne sont pas implémentables

On peut toujours les spécifier et vérifier le code client

```
type array 'a model { mutable elts: map int 'a;
                                length: int }
invariant { 0 <= self.length }
```

- les champs `length` et `elts` ne sont utilisables que dans des annotations logiques (type modèle)
- tout accès est fait via des fonctions abstraites
- l'invariant est vérifié à l'entrée et à la sortie des fonctions
  - WHY3 ajoute pré- et postconditions implicites

Spécification abstraites

```
type array 'a model { mutable elts: map int 'a;
                                length: int }
invariant { 0 <= self.length }

val ([]) (a: array 'a) (i: int): 'a
  requires { 0 <= i < a.length }
  ensures  { result = a.elts[i] }

val ([]<-) (a: array 'a) (i: int) (v: 'a): unit writes {a}
  requires { 0 <= i < a.length }
  ensures  { a.elts = (old a.elts)[i <- v] }

val length (a: array 'a): int ensures { result = a.length }

function get (a: array 'a) (i: int): 'a = a.elts[i]
```

- les champs immuables sont préservés — postcondition implicite
- la fonction logique `get` n'a pas de précondition
  - son résultat en dehors de bornes n'est pas spécifié

4. Exercices

Exercice 1 : arbres binaires

```
type tree 'a = Leaf | Node (tree 'a) 'a (tree 'a)
```

Spécifier, implémenter et vérifier une fonction qui calcule le maximum dans un arbre d'entiers.

problème 2 de la compétition de vérification FoVeOOS en 2011,
http://foveoos2011.cost-ic0701.org/verification-competition

Exercice 2 : règles de WP

Proposer des règles de calcul de la plus faible précondition :

- pour le « et » paresseux qui évalue la 2e partie que si la 1re est vraie

$$e_1 \,\&\&\, e_2$$

- pour une conditionnelle où la condition est une expression

$$\texttt{if } e_c \texttt{ then } e_1 \texttt{ else } e_2$$

- pour une boucle `while` où la condition est une expression

$$\texttt{while } e_1 \texttt{ invariant } J \texttt{ do } e_2 \texttt{ done}$$