Vérification modulaire d'invariants

Sylvain Boulmé et Marie-Laure Potet

Juin 2008

Plan

Certification de Programmes Modulaires et Impératifs

ARC INRIA démarré en mars 2008.

Objectifs Vérifier programmes avec assistants de preuves :

- modularité, invariants, raffinements.
- partage d'états entre modules, alias.
- ordre supérieur.

Membres et technos dans CeProMi

- C. Marché, JC. Filliatre, etc (INRIA ProVal) : platef. Why.
  Entrée : ML, Java, C avec annot. à la Hoare.
  Sortie : OP pour prouveurs SMT-LIB dont Ergo.
- F. Pottier, A. Charégaud (INRIA Gallium) : Capacités.
  Typage inspiré des effets/régions & de logique de séparation.
  $\Rightarrow$ sdd mutables + ordre sup (système F)
  Traduction fonctionnelle vers Coq : preuve sur pg traduit.
- S. Boulmé, M.L. Potet (Verimag DCS) : méthode B & DSM.
  Dijkstra Specification Monads :
  - raffinement algorithmique en Coq avec ordre sup.
  - calculs de WP = Curry-Howard pour effets.

  méthode B : composition, invariants et raffinements.
- A. Giorgetti, etc (INRIA Cassis) : méthode B (& Harvey ?).

Idée forte de la méthode B

Méthode formelle de la conception jusqu'à l'implémentation dans une démarche "top-down" (formellement : le raffinement).

Idée forte de la méthode B

Méthode formelle de la conception jusqu'à l'implémentation dans une démarche "top-down" (formellement : le raffinement).

Intérêts du raffinement :

- formalisation hiérarchique du cahier des charges
  1. propriétés "fondamentales" du système (fonction de base).
  2. propriétés "secondaires" (moyens à utiliser, architecture, etc). Le raffinement garantit que ce niveau ne viole pas les propriétés du niveau précédent.
  3. itération jusqu'au code.
- méthode très efficace pour faire apparaître le "non-dit" et "l'arbitraire" dans le cahier des charges.
- une grande traçabilité des exigences informelles du cahier des charges.
- une certaine adaptabilité au changement.
- abstraction et simplification des preuves.

Langage B

Une hiérarchie de langages :

1. 1 logique du 1er ordre + 1 théorie des ensembles (typée)
2. substitution généralisée (raffinement algorithmique sur instructions).
3. machines B (composants, raffinement de composants).
   2 versions de ce niveau : B procédural ou B événementiel.

Langage B

Une hiérarchie de langages :

1. 1 logique du 1er ordre + 1 théorie des ensembles (typée)
2. substitution généralisée (raffinement algorithmique sur instructions).
3. machines B (composants, raffinement de composants).
   2 versions de ce niveau : B procédural ou B événementiel.

Composition avec partage d'état (très) limité :

- un seul écrivain.
- restrictions sur les lecteurs (état partagé pas dans l'invariant).

Introduction

Nos travaux en bref

- expliquer la correction de la notion d'invariant en B (en s'inspirant de l'approche Spec#).
- extension conservative de B avec modules plus expressifs.

Introduction

Nos travaux en bref

- expliquer la correction de la notion d'invariant en B (en s'inspirant de l'approche Spec#).
- extension conservative de B avec modules plus expressifs.

Objectifs de l'exposé

- expliquer notre approche informellement dans un cadre plus général que B (en restant dans le cadre "absence d'alias").
- discuter de qq problèmes sur la vérif. d'invariants (approche généralisable avec des régions ou capacités ? ? ?)

Plan

1er objectif : limiter "naturellement" le nombre d'OP

```
let foo (...) = { P ∧ I } ... { Q ∧ I }
```

1er objectif : limiter "naturellement" le nombre d'OP

```
let foo (...) = { P ∧ I } ... { Q ∧ I }
```

Règle usuelle de l'appel

$$\frac{P' \Rightarrow (P \wedge I) \quad (Q \wedge I) \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{foo}(\ldots)\ \{\ Q'\ \}}$$

1er objectif : limiter "naturellement" le nombre d'OP

```
let foo (...) = { P ∧ I } ... { Q ∧ I }
```

Règle usuelle de l'appel

$$\frac{P' \Rightarrow (P \wedge I) \quad (Q \wedge I) \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{foo}(\ldots)\ \{\ Q'\ \}}$$

On veut pouvoir utiliser cette règle à la place

$$\frac{\text{“}I \text{ invariant valide dans le contexte”} \quad (P' \wedge I) \Rightarrow P \quad (Q \wedge I) \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{foo}(\ldots)\ \{\ Q'\ \}}$$

1er objectif : limiter "naturellement" le nombre d'OP

```
let foo (...) = { P ∧ I } ... { Q ∧ I }
```

Règle usuelle de l'appel

$$\frac{P' \Rightarrow (P \wedge I) \quad (Q \wedge I) \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{foo}(...)\ \{\ Q'\ \}}$$

On veut pouvoir utiliser cette règle à la place

$$\frac{\text{"}I \text{ invariant valide dans le contexte"} \quad (P' \wedge I) \Rightarrow P \quad (Q \wedge I) \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{foo}(...)\ \{\ Q'\ \}}$$

Difficulté : définir et vérifier la "validité" des invariants.

- invariant sur plus de 2 variables $\Rightarrow$ pas vrai tt le temps.
- on se base sur :

$$\frac{\{\ P_0\ \}\ e\ \{\ Q_0\ \} \quad \text{"}e \text{ ne modifie pas les var. de } FV(I)\text{"}}{\{\ P_0 \wedge I\ \}\ e\ \{\ Q_0 \wedge I\ \}}\ \textbf{FRAME}$$

- **Notion d'appartenance** : si I considéré valide, var ds $FV(I)$ non modifiables.

Objectif proche : abstraction/raffinement de données

```
let a = ref ... ;;
op: ... { P(a) } writes a { Q(a@,a) }
```

Objectif proche : abstraction/raffinement de données

```
let a = ref ... ;;
op: ... { P(a) } writes a { Q(a@,a) }

let c = ref ... ;; { inv de collage: I(a,c) }
let op ... = e (* writes c *) ;;
```

Objectif proche : abstraction/raffinement de données

```
let a = ref ... ;;
op: ... { P(a) } writes a { Q(a@,a) }

let c = ref ... ;; { inv de collage: I(a,c) }
let op ... = e (* writes c *) ;;
```

Au moment de la définition

$$\{\ P(a) \wedge I(a,c)\ \}\ e\ \{\ \exists a'.I(a',c) \wedge Q(a,a')\ \}$$

Règle à l'appel

$$\frac{\text{“I valide dans le contexte”} \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{op}(\ldots)\ \{\ Q'\ \}}$$

Objectif proche : abstraction/raffinement de données

```
let a = ref ... ;;
op: ... { P(a) } writes a { Q(a@,a) }

let c = ref ... ;; { inv de collage: I(a,c) }
let op ... = e (* writes c *) ;;
```

Au moment de la définition

$$\{\ P(a) \wedge I(a,c)\ \}\ e\ \{\ \exists a'.I(a',c) \wedge Q(a,a')\ \}$$

Règle à l'appel

$$\frac{\text{“I valide dans le contexte”} \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{op}(\ldots)\ \{\ Q'\ \}}$$

Ici : on veut gérer les invariants “cachés”.

2-ième objectif : hiérarchie d'invariants

```
module A =
  struct
    let x = ref 0 { INVARIANT x >= 0 }
    let incr () = x:=!x+1
  end

module B =
  struct
    let y = ref (2 * !A.x)  { INVARIANT y=2*A.x }
    let foo () = y:=!y+2 ; A.incr()
  end
```

Règle : invariant de B valide $\Rightarrow$ invariant de A valide.

2-ième objectif : hiérarchie d'invariants

```
module A =
  struct
    let x = ref 0 { INVARIANT x >= 0 }
    let incr () = x:=!x+1
  end

module B =
  struct
    let y = ref (2 * !A.x)  { INVARIANT y=2*A.x }
    let foo () = y:=!y+2 ; A.incr()
  end
```

Règle : invariant de B valide $\Rightarrow$ invariant de A valide.

Avantages :

- simplifie la compréhension de quels invariants sont valides.
- nécessaire en présence de raffinement de données.

3-ième objectif : modularité + invariants + partage

Patron : un écrivain / des lecteurs

- un module A avec état.
- un module *Ecrivain* qui lit et écrit dans A et dont l'invariant porte sur l'état de A.
- lecteurs : des modules qui lisent dans A et dont l'invariant porte sur l'état de A.

3-ième objectif : modularité + invariants + partage

Patron : un écrivain / des lecteurs

- un module A avec état.
- un module *Ecrivain* qui lit et écrit dans A et dont l'invariant porte sur l'état de A.
- lecteurs : des modules qui lisent dans A et dont l'invariant porte sur l'état de A.

Exemples :

- une BD, avec un module d'administration et un module d'utilisation.
- un module A, un module d'initialisation de A et un autre d'utilisation de A.

3-ième objectif : modularité + invariants + partage

Patron : un écrivain / des lecteurs

- un module A avec état.
- un module *Ecrivain* qui lit et écrit dans A et dont l'invariant porte sur l'état de A.
- lecteurs : des modules qui lisent dans A et dont l'invariant porte sur l'état de A.

Exemples :

- une BD, avec un module d'administration et un module d'utilisation.
- un module A, un module d'initialisation de A et un autre d'utilisation de A.

Restriction de B : pas d'invariants sur A dans les lecteurs.

3-ième objectif : modularité + invariants + partage

Patron : un écrivain / des lecteurs

- un module A avec état.
- un module *Ecrivain* qui lit et écrit dans A et dont l'invariant porte sur l'état de A.
- lecteurs : des modules qui lisent dans A et dont l'invariant porte sur l'état de A.

Exemples :

- une BD, avec un module d'administration et un module d'utilisation.
- un module A, un module d'initialisation de A et un autre d'utilisation de A.

Restriction de B : pas d'invariants sur A dans les lecteurs.
Ici, on va considérer le cas de **plusieurs écrivains**.

Ex 1 : 2 écrivains avec invariants pas tjs valides

```
module A = struct
  let x = ref 0 { INVARIANT x >= 0 }
  let incr () = x:=!x+1
end

module B = struct
  let y = ref (2 * !A.x)  { INVARIANT y=2*A.x }
  let reset () = y:=2 * !A.x
  let foo () = y:=!y+2 ; A.incr()
end

module C = struct
  let y = ref (2 * !A.x + 1) { INVARIANT y=2*A.x+1 }
  let reset () = y:=(2 * !A.x + 1)
  let foo () = y:=!y+2 ; A.incr() ;
end
```

Ex 1 : 2 écrivains avec invariants pas tjs valides

```
module A = struct
  let x = ref 0 { INVARIANT x >= 0 }
  let incr () = x:=!x+1
end

module B = struct
  let y = ref (2 * !A.x)  { INVARIANT y=2*A.x }
  let reset () = y:=2 * !A.x
  let foo () = y:=!y+2 ; A.incr()
end

module C = struct
  let y = ref (2 * !A.x + 1) { INVARIANT y=2*A.x+1 }
  let reset () = y:=(2 * !A.x + 1)
  let foo () = y:=!y+2 ; A.incr() ;
end
```

La séquence `B.foo() ; C.foo()` doit être rejetée.

Seules certaines séquences sont acceptables :

```
B.reset() ; B.foo() ; B.foo() ; C.reset() ; C.foo() ; C.foo() ;
```

Ex 2 : écrivains avec invariants tjs valides

```
module Buff = struct
  let l = ref []
  let prem = ref 0 { INVARIANT 0 <= prem <= List.length l }
  let add x = l:= !l @ [x]
  let remove () =
      if !prem < List.length !l then
        let v=List.nth !l !prem in prem:=!prem+1 ; v
      else 0
end

module Producer = struct
  let next = ref 1
   { INVARIANT Buff.l = liste des ent. entre 1 et (next-1) }
  let step () = Buff.add !next ; next:=!next+1
end

module Consumer = struct
  let s = ref 0
      { INVARIANT s = somme des elts de la sous-liste de
          Buff.l entre indice 0 et indice (Buff.prem-1) }
  let step () = s:=!s + Buff.remove()
end
```

Plan

Introduction

- **règle de l'appel avec invariant** :

$$\frac{\text{“}I \text{ invariant valide dans le contexte”} \quad (P' \wedge I) \Rightarrow P \quad (Q \wedge I) \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{foo}(\ldots)\ \{\ Q'\ \}}$$

Ici, la vérification que “I invariant valide dans le contexte” se fait par preuve.

Introduction

- **règle de l'appel avec invariant** :

$$\frac{\text{“}I \text{ invariant valide dans le contexte”} \quad (P' \wedge I) \Rightarrow P \quad (Q \wedge I) \Rightarrow Q'}{\{\ P'\ \}\ \mathrm{foo}(\ldots)\ \{\ Q'\ \}}$$

Ici, la vérification que “I invariant valide dans le contexte” se fait par preuve.

- Notion d'invariant basé sur un système d'appartenance. (orthogonale à la notion de modules, d'encapsulation, etc).

- “Méta-modèle” (utilisation de variables fantômes).

Références et Invariants

Chaque référence x porte un invariant nommé $\mathrm{Inv}(x)$.
Exemple :

```
let l = ref [] { INVARIANT True }
let prem = ref 0
  { INVARIANT prem. 0 <= prem <= List.length l }
```

est très similaire de

```
let prem = ref 0 { INVARIANT True }
let l = ref [] { INVARIANT l. 0 <= prem <= List.length l }
```

qui est proche de

```
let prem = ref 0 { INVARIANT True }
let l = ref [] { INVARIANT True }
let dummy = ref () { INVARIANT 0 <= prem <= List.length l }
```

Admissibilité des invariants

Appartenance statique : relation `owns` "sans cycle" (i.e. dont la cloture transitive $\texttt{owns}^+$ est irreflexive).

Admissibilité de $\mathrm{Inv}(x)$ vis-à-vis de `owns`

$$FV(\mathrm{Inv}(x)) \subseteq \texttt{owns}^*(\{x\})$$

Ici, on suppose `owns` donné par utilisateur (voir exemples plus loin).

Statut des invariants

Chaque référence est vue comme un record à 2 champs mutables :

- $x.c$ le "contenu" de la ref.
- $x.s$ le "statut" à valeur ds $\{\texttt{invalid}, \texttt{valid}, \texttt{comm}\}$.
  Champ non modifiable explicitement.

Appartenance dynamique :
x propriétaire dynamique de y ssi x `owns` y et $x.s \notin \texttt{invalid}$.

Signification de $x.s$:

- $x.s = \texttt{invalid}$ signifie que $\mathrm{Inv}(x)$ n'est pas garantit.
- $x.s = \texttt{valid}$ signifie que $\mathrm{Inv}(x)$ est garanti et sans propriétaire
- $x.s = \texttt{comm}$ signifie que $\mathrm{Inv}(x)$ est garanti et a un unique propriétaire

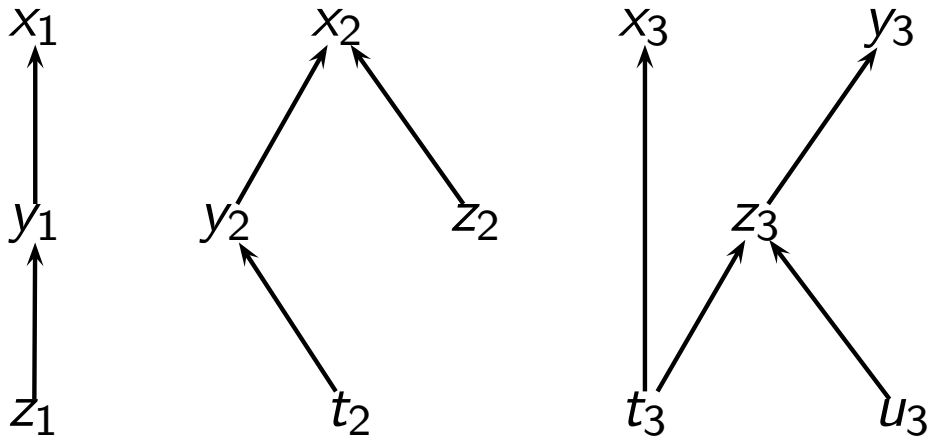
Méta-invariant

Pour toutes références x et y,

$\mathcal{MI}_1$	$x.s \neq \texttt{invalid} \Rightarrow \mathrm{Inv}(x)$
$\mathcal{MI}_2$	$x.s \neq \texttt{invalid} \ \wedge \ x \ \texttt{owns} \ y \Rightarrow y.s = \texttt{comm}$
$\mathcal{MI}_3$	$x.s \neq \texttt{invalid} \ \wedge \ \texttt{owns}(\{x\}) \cap \texttt{owns}(\{y\}) \neq \emptyset \ \wedge \ x \neq y \Rightarrow y.s = \texttt{invalid}$

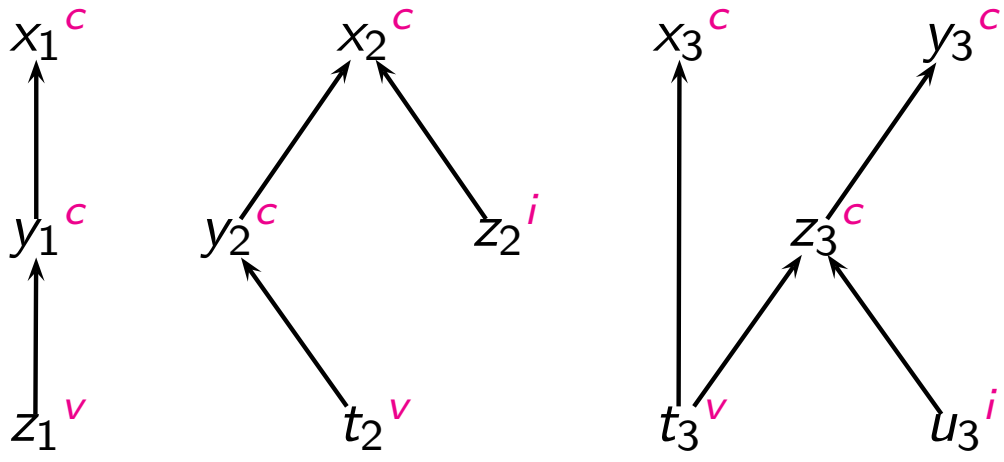
Contraintes de $\mathcal{MI}$ sur owns

Invariants tous validables.



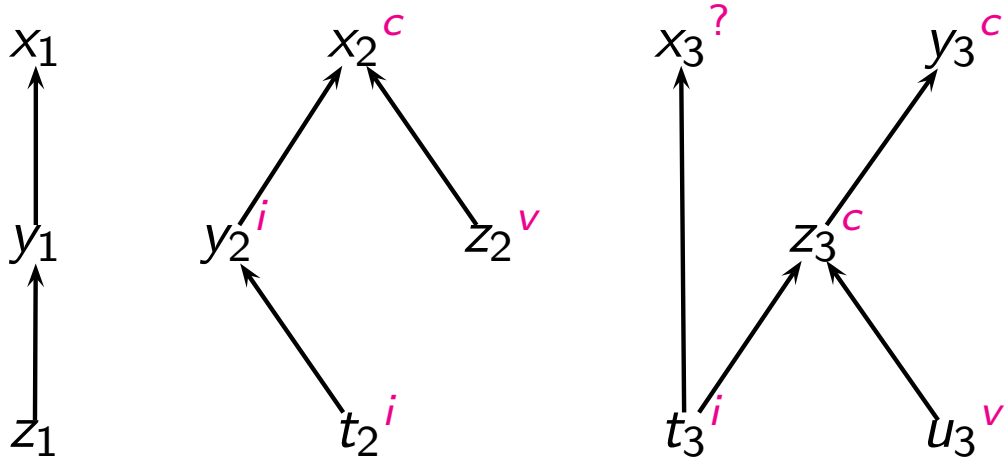
Contraintes de $\mathcal{MI}$ sur owns

Invariants tous validables.



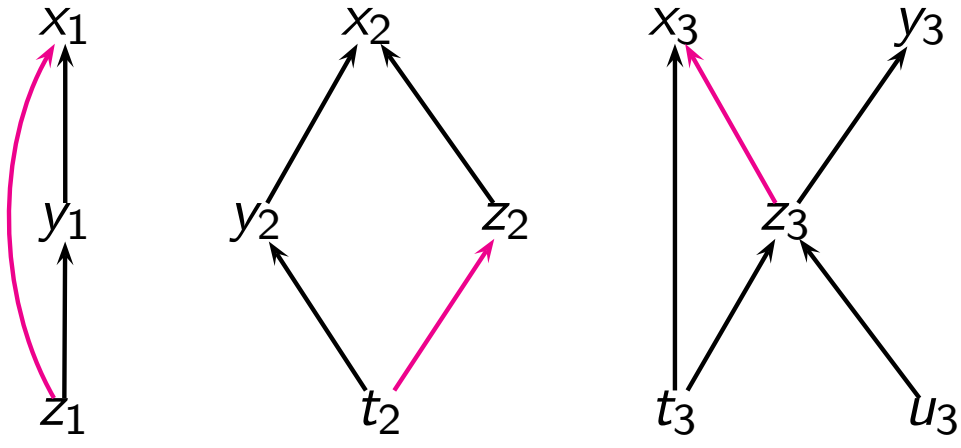
Contraintes de $\mathcal{MI}$ sur owns

Invariants tous validables.



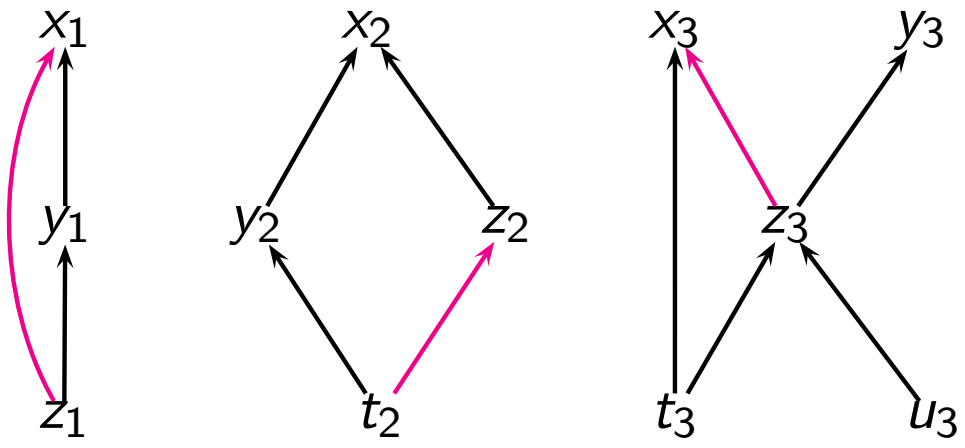
Contraintes de $\mathcal{MI}$ sur `owns`

Invariants pas tous validables.



Contraintes de $\mathcal{MI}$ sur owns

Invariants pas tous validables.

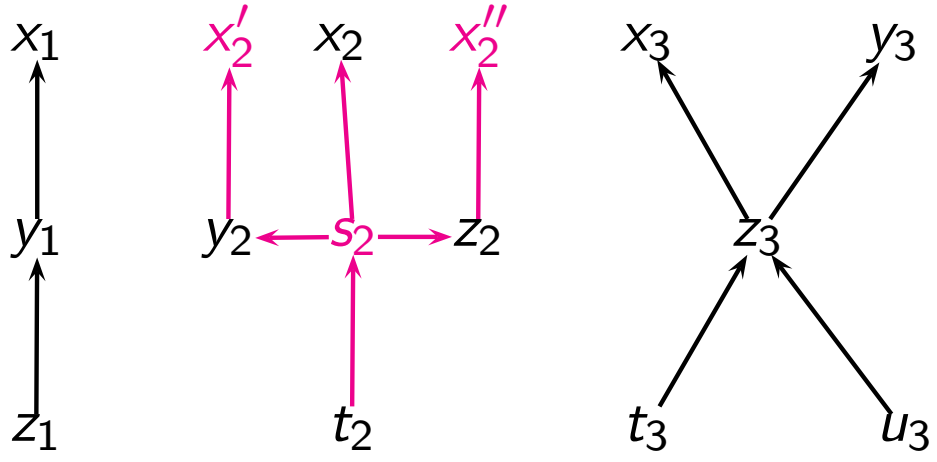


En pratique :

- "triangles" (archi 1 et 3) évitables.

Contraintes de $\mathcal{MI}$ sur owns

Invariants tous validables.



En pratique :

- "triangles" (archi 1 et 3) évitables.
- Pour "diamants" (archi 2), on peut dupliquer x_2 et ajouter un invariant (nommé s_2) intermédiaire $x_2 = x_2' \wedge x_2 = x_2''$.

Décoration

Affectation pré-conditionnée + ajout de 2 instructions pack et unpack qui ne font que lirent et écrirent dans les statuts.

$$
\begin{aligned}
wp(x := e, Q) &= x.s = \texttt{invalid} \\
&\quad \wedge\ Q[x.c \leftarrow e] \\
wp(\texttt{pack}(x), Q) &= (\wedge_{y \in \texttt{owns}(\{x\})} y.s = \texttt{valid}) \\
&\quad \wedge\ \mathrm{Inv}(x) \\
&\quad \wedge\ Q[y.s \leftarrow \texttt{comm}]_{y \in \texttt{owns}(\{x\})}[x.s \leftarrow \texttt{valid}] \\
wp(\texttt{unpack}(x), Q) &= x.s = \texttt{valid} \\
&\quad \wedge\ Q[y.s \leftarrow \texttt{valid}]_{y \in \texttt{owns}(\{x\})}[x.s \leftarrow \texttt{invalid}]
\end{aligned}
$$

Décoration

Affectation pré-conditionnée + ajout de 2 instructions pack et unpack qui ne font que lirent et écrirent dans les statuts.

$$
\begin{aligned}
wp(x := e, Q) &= x.s = \texttt{invalid} \\
&\quad \wedge\ Q[x.c \leftarrow e] \\
wp(\texttt{pack}(x), Q) &= (\wedge_{y \in \texttt{owns}(\{x\})} y.s = \texttt{valid}) \\
&\quad \wedge\ \mathrm{Inv}(x) \\
&\quad \wedge\ Q[y.s \leftarrow \texttt{comm}]_{y \in \texttt{owns}(\{x\})}[x.s \leftarrow \texttt{valid}] \\
wp(\texttt{unpack}(x), Q) &= x.s = \texttt{valid} \\
&\quad \wedge\ Q[y.s \leftarrow \texttt{valid}]_{y \in \texttt{owns}(\{x\})}[x.s \leftarrow \texttt{invalid}]
\end{aligned}
$$

Thm pour tout S, $(\mathcal{MI} \wedge wp(S, True)) \Rightarrow wp(S, \mathcal{MI})$.

Ex 1 : décorations

```
module A = struct
  let x = ref 0 { INVARIANT x >= 0 }
  let incr () =  x:=!x+1
end

module B = struct
  let y = ref (2 * !A.x)  { INVARIANT y=2*A.x  }
  let reset () = y:= 2 * !A.x
  let foo () =  y:=!y+2 ; A.incr()
end

module C = struct
  let y = ref (2 * !A.x + 1) { INVARIANT y=2*A.x+1  }
  let reset () = y:= 2 * !A.x + 1
  let foo () =  y:=!y+2 ; A.incr()
end
```

Ex 1 : décorations

```
module A = struct
  let x = ref 0 { INVARIANT x >= 0 }
  let incr () = unpack(x); x:=!x+1 ; pack(y)
end

module B = struct
  let y = ref (2 * !A.x)  { INVARIANT y=2*A.x  }
  let reset () = y:= 2 * !A.x
  let foo () =  y:=!y+2 ; A.incr()
end

module C = struct
  let y = ref (2 * !A.x + 1) { INVARIANT y=2*A.x+1  }
  let reset () = y:= 2 * !A.x + 1
  let foo () =  y:=!y+2 ; A.incr()
end
```

Ex 1 : décorations

```
module A = struct
  let x = ref 0 { INVARIANT x >= 0 }
  let incr () = unpack(x); x:=!x+1 ; pack(y)
end

module B = struct
  let y = ref (2 * !A.x)  { INVARIANT y=2*A.x OWNS A.x }
  let reset () = y:= 2 * !A.x
  let foo () =  y:=!y+2 ; A.incr()
end

module C = struct
  let y = ref (2 * !A.x + 1) { INVARIANT y=2*A.x+1  }
  let reset () = y:= 2 * !A.x + 1
  let foo () =  y:=!y+2 ; A.incr()
end
```

Ex 1 : décorations

```
module A = struct
  let x = ref 0 { INVARIANT x >= 0 }
  let incr () = unpack(x); x:=!x+1 ; pack(y)
end

module B = struct
  let y = ref (2 * !A.x)  { INVARIANT y=2*A.x OWNS A.x }
  let reset () = y:= 2 * !A.x ; pack(y)
  let foo () =  y:=!y+2 ; A.incr()
end

module C = struct
  let y = ref (2 * !A.x + 1) { INVARIANT y=2*A.x+1  }
  let reset () = y:= 2 * !A.x + 1
  let foo () =  y:=!y+2 ; A.incr()
end
```

Ex 1 : décorations

```
module A = struct
  let x = ref 0 { INVARIANT x >= 0 }
  let incr () = unpack(x); x:=!x+1 ; pack(y)
end

module B = struct
  let y = ref (2 * !A.x)  { INVARIANT y=2*A.x OWNS A.x }
  let reset () = y:= 2 * !A.x ; pack(y)
  let foo () = unpack(y); y:=!y+2 ; A.incr() ; pack(y)
end

module C = struct
  let y = ref (2 * !A.x + 1) { INVARIANT y=2*A.x+1  }
  let reset () = y:= 2 * !A.x + 1
  let foo () =  y:=!y+2 ; A.incr()
end
```

Ex 1 : décorations

```
module A = struct
  let x = ref 0 { INVARIANT x >= 0 }
  let incr () = unpack(x); x:=!x+1 ; pack(y)
end

module B = struct
  let y = ref (2 * !A.x)  { INVARIANT y=2*A.x OWNS A.x }
  let reset () = y:= 2 * !A.x ; pack(y)
  let foo () = unpack(y); y:=!y+2 ; A.incr() ; pack(y)
end

module C = struct
  let y = ref (2 * !A.x + 1) { INVARIANT y=2*A.x+1 OWNS A.x }
  let reset () = y:= 2 * !A.x + 1 ; pack(y)
  let foo () = unpack(y); y:=!y+2 ; A.incr() ; pack(y)
end
```

Ex 1 : refuser la séquence 1

```
module A : sig
  x : int ref { INVARIANT ... }
  incr : unit ->
    { x.s=valid } unit writes x { x.s=valid }
end

module B,C : sig
  y: int ref { INVARIANT ... OWNS A.x }
  reset : unit ->
    { A.x.s=valid } unit reads A.x writes y { y.s = valid }
  foo: unit ->
    { y.s = valid } unit writes y,A.x  { y.s = valid }
end
```

Ex 1 : refuser la séquence 1

```
module A : sig
  x : int ref { INVARIANT ... }
  incr : unit ->
    { x.s=valid } unit writes x { x.s=valid }
end

module B,C : sig
  y: int ref { INVARIANT ... OWNS A.x }
  reset : unit ->
    { A.x.s=valid } unit reads A.x writes y { y.s = valid }
  foo: unit ->
    { y.s = valid } unit writes y,A.x  { y.s = valid }
end
```

La séquence suivante est incorrecte (i.e. cette séquence S vérifie $(\mathcal{MI} \wedge wp(S, True)) \Rightarrow False$) :

```
B.foo () ; C.foo() ;
```

En effet, $B.y.s = \texttt{valid} \wedge \mathcal{MI} \Rightarrow wp(\texttt{B.foo()}, C.y.s = \texttt{invalid})$.

Ex 1 : accepter la séquence 2

```
module B,C : sig
  y: int ref { INVARIANT ... OWNS A.x }
  reset : unit ->
    { A.x.s=valid } unit reads A.x writes y { y.s = valid }
  foo: unit ->
    { y.s = valid } unit writes y,A.x  { y.s = valid }
end
```

- La séquence suivante est aussi incorrecte :

```
B.foo() ; B.foo() ; C.reset() ; C.foo() ; C.foo() ;
```

En effet, $B.y.s = \texttt{valid} \wedge \mathcal{MI} \Rightarrow wp(\texttt{B.foo()}, A.x.s = \texttt{comm})$

Ex 1 : accepter la séquence 2

```
module B,C : sig
  y: int ref { INVARIANT ... OWNS A.x }
  reset : unit ->
    { A.x.s=valid } unit reads A.x writes y { y.s = valid }
  foo: unit ->
    { y.s = valid } unit writes y,A.x  { y.s = valid }
end
```

- La séquence suivante est aussi incorrecte :

```
B.foo() ; B.foo() ; C.reset() ; C.foo() ; C.foo() ;
```

En effet, $B.y.s = \texttt{valid} \wedge \mathcal{MI} \Rightarrow wp(\texttt{B.foo()}, A.x.s = \texttt{comm})$

- Séquence suivante ok (avec précondition $B.y.s = \texttt{valid} \wedge \mathcal{MI}$) :

```
B.foo() ; B.foo() ; unpack(B.y) ; C.reset() ; C.foo() ; C.foo() ;
```

Ex 2 : pb. d'archi

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l }
  ...

module Producer = struct
  let next = ref 1
    { INVARIANT Buff.l = liste des ent. entre 1 et (next-1)
      }
  ...

module Consumer = struct
  let s = ref 0
    { INVARIANT s = somme des elts de la sous-liste de
      Buff.l entre indice 0 et indice (Buff.prem-1)
      }
  ...
```

Ex 2 : pb. d'archi

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l}
  ...

module Producer = struct
  let next = ref 1
    { INVARIANT Buff.l = liste des ent. entre 1 et (next-1)
      }
  ...

module Consumer = struct
  let s = ref 0
    { INVARIANT s = somme des elts de la sous-liste de
      Buff.l entre indice 0 et indice (Buff.prem-1)
      }
  ...
```

Ex 2 : pb. d'archi

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l}
  ...

module Producer = struct
  let next = ref 1
    { INVARIANT Buff.l = liste des ent. entre 1 et (next-1)
      OWNS Buff.l }
  ...

module Consumer = struct
  let s = ref 0
    { INVARIANT s = somme des elts de la sous-liste de
      Buff.l entre indice 0 et indice (Buff.prem-1)
       }
  ...
```

Ex 2 : pb. d'archi

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l}
  ...

module Producer = struct
  let next = ref 1
    { INVARIANT Buff.l = liste des ent. entre 1 et (next-1)
      OWNS Buff.l }
  ...

module Consumer = struct
  let s = ref 0
    { INVARIANT s = somme des elts de la sous-liste de
      Buff.l entre indice 0 et indice (Buff.prem-1)
       }
  ...
```

Pb : `Producer.next` et `Buff.prem` jamais valides en même temps

Ex 2 : pb. d'archi

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l}
  ...

module Producer = struct
  let next = ref 1
    { INVARIANT Buff.l = liste des ent. entre 1 et (next-1)
      OWNS Buff.prem }
  ...

module Consumer = struct
  let s = ref 0
    { INVARIANT s = somme des elts de la sous-liste de
      Buff.l entre indice 0 et indice (Buff.prem-1)
       }
  ...
```

Ex 2 : pb. d'archi

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l}
  ...

module Producer = struct
  let next = ref 1
    { INVARIANT Buff.l = liste des ent. entre 1 et (next-1)
      OWNS Buff.prem }
  ...

module Consumer = struct
  let s = ref 0
    { INVARIANT s = somme des elts de la sous-liste de
      Buff.l entre indice 0 et indice (Buff.prem-1)
      OWNS Buff.prem }
  ...
```

Ex 2 : pb. d'archi

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l}
  ...

module Producer = struct
  let next = ref 1
    { INVARIANT Buff.l = liste des ent. entre 1 et (next-1)
      OWNS Buff.prem }
  ...

module Consumer = struct
  let s = ref 0
    { INVARIANT s = somme des elts de la sous-liste de
      Buff.l entre indice 0 et indice (Buff.prem-1)
      OWNS Buff.prem }
  ...
```

Pb : `Producer.next` et `Consumer.s` jamais valides en même temps

Ex 2 : une bonne archi

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l }
  ...

module Producer = struct
  let next = ref 1
  let l = ref []
    { INVARIANT l = liste des ent. entre 1 et (next-1)
      OWNS next }
  ...

module Consumer = struct
  let s = ref 0
  let l = ref [] { INVARIANT s = somme des elts de l OWNS s }
  ...

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Ex 2 : décoration de Buff

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l }
  let add x =


      l:= !l @ [x]


  let remove () =
      if !prem < List.length !l then
        (
         let v=List.nth !l !prem in prem:=!prem+1 ;

           v)
      else 0
end
```

Ex 2 : décoration de Buff

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l }
  let add x =
      unpack(prem);
      unpack(l); (* inutile si optim. des inv. triviaux *)
      l:= !l @ [x]
      pack(l); (* inutile si optim. des inv. triviaux *)
      pack(prem)
  let remove () =
      if !prem < List.length !l then
        (
         let v=List.nth !l !prem in prem:=!prem+1 ;

           v)
      else 0
end
```

Ex 2 : décoration de Buff

```
module Buff = struct
  let l = ref []
  let prem = ref 0
    { INVARIANT 0 <= prem <= List.length l OWNS l }
  let add x =
      unpack(prem);
      unpack(l); (* inutile si optim. des inv. triviaux *)
      l:= !l @ [x]
      pack(l); (* inutile si optim. des inv. triviaux *)
      pack(prem)
  let remove () =
      if !prem < List.length !l then
        (unpack(prem);
         let v=List.nth !l !prem in prem:=!prem+1 ;
           pack(prem);
           v)
      else 0
end
```

Ex 2 : décoration de Producer

```
module Producer = struct
  let next = ref 1
  let l = ref []
    { INVARIANT l = liste des ent. entre 1 et (next-1)
      OWNS next }
  let step () =

    Buff.add !next ;   (* ici: Buff.prem=valid *)


    next:=!next+1




end

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Ex 2 : décoration de Producer

```
module Producer = struct
  let next = ref 1
  let l = ref []
    { INVARIANT l = liste des ent. entre 1 et (next-1)
      OWNS next }
  let step () =
    unpack(market);
    Buff.add !next ;   (* ici: Buff.prem=valid *)


    next:=!next+1



    pack(market) (* ici : prop sur Consumer.l tjs ok *)
end

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Ex 2 : décoration de Producer

```
module Producer = struct
  let next = ref 1
  let l = ref []
    { INVARIANT l = liste des ent. entre 1 et (next-1)
      OWNS next }
  let step () =
    unpack(market);
    Buff.add !next ;   (* ici: Buff.prem=valid *)
    unpack(l);
    unpack(next); (* inutile si optim. des inv. triviaux *)
    next:=!next+1
    pack(next); (* inutile si optim. des inv. triviaux *)

    pack(l);
    pack(market) (* ici : prop sur Consumer.l tjs ok *)
end

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Ex 2 : décoration de Producer

```
module Producer = struct
  let next = ref 1
  let l = ref []
    { INVARIANT l = liste des ent. entre 1 et (next-1)
      OWNS next }
  let step () =
    unpack(market);
    Buff.add !next ;   (* ici: Buff.prem=valid *)
    unpack(l);
    unpack(next); (* inutile si optim. des inv. triviaux *)
    next:=!next+1
    pack(next); (* inutile si optim. des inv. triviaux *)
    l :=Buff.l;
    pack(l);
    pack(market) (* ici : prop sur Consumer.l tjs ok *)
end

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Ex 2 : décoration de Consumer

```
module Consumer = struct
  let s = ref 0
  let l = ref [] { INVARIANT s = somme des elts de l OWNS s }
  let step () =



    s := !s + Buff.remove()




end

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Ex 2 : décoration de Consumer

```
module Consumer = struct
  let s = ref 0
  let l = ref [] { INVARIANT s = somme des elts de l OWNS s }
  let step () =
    unpack(market);


    s := !s + Buff.remove()



    pack(market) (* ici : prop sur Producer.l tjs ok *)
end

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Ex 2 : décoration de Consumer

```
module Consumer = struct
  let s = ref 0
  let l = ref [] { INVARIANT s = somme des elts de l OWNS s }
  let step () =
    unpack(market);
    unpack(l);
    unpack(s); (* inutile si optim. des inv. triviaux *)
    s := !s + Buff.remove()
    pack(s); (* inutile si optim. des inv. triviaux *)

    pack(l);
    pack(market) (* ici : prop sur Producer.l tjs ok *)
end

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Ex 2 : décoration de Consumer

```
module Consumer = struct
  let s = ref 0
  let l = ref [] { INVARIANT s = somme des elts de l OWNS s }
  let step () =
    unpack(market);
    unpack(l);
    unpack(s); (* inutile si optim. des inv. triviaux *)
    s := !s + Buff.remove()
    pack(s); (* inutile si optim. des inv. triviaux *)
    l := "ss-liste de Buff.l ...";
    pack(l);
    pack(market) (* ici : prop sur Producer.l tjs ok *)
end

let market = ref ()
  { INVARIANT Producer.l=Buff.l  and  Consumer.l="ss-liste
                de Buff.l entre ind 0 et (Buff.prem-1)"
    OWNS Producer.l, Consumer.l, Buff.prem }
```

Conclusion de cette approche

Avantages

- expressivité
- indépendant des mécanismes de modularité et d'encapsulation.

Inconvénients

- lourd dans les annotations (pré/post) et dans les OP.
- difficulté liée aux risques d'inconsistence (i.e. pb d'archi).

Plan

Objectifs

- l'utilisateur écrit les `unpack` et les `pack`.
- l'analyseur infère et vérifie les annotations (pré/post) sur les statuts.
- l'analyseur génère les OP (invariants en hypothèse et en conclusions).
- principale restriction d'expressivité : statut des invariants indépendant du contrôle.

Objectifs

- l'utilisateur écrit les `unpack` et les `pack`.
- l'analyseur infère et vérifie les annotations (pré/post) sur les statuts.
- l'analyseur génère les OP (invariants en hypothèse et en conclusions).
- principale restriction d'expressivité : statut des invariants indépendant du contrôle.

Exemple La ligne suivante est rejetée :

```
if ... then pack(x)
```

Tandis que la suivante est ok :

```
if ... then unpack(x); x:=x+1; pack(x)
```

Annotations décidables sur les statuts

Logique $\mathcal{L}_{\mathrm{S}}$: logique propositionnelle classique avec atomes de la forme "$x.\mathrm{S} \in A$" avec $A \subseteq \{\texttt{invalid}, \texttt{valid}, \texttt{comm}\}$.
Ici : égalité syntaxique sur les variables (i.e. absence d'alias).

Propriété 1 : $\mathcal{L}_{\mathrm{S}}$ décidable.

Annotations décidables sur les statuts

Logique $\mathcal{L}_{\mathrm{s}}$: logique propositionnelle classique avec atomes de la forme "$x.\mathrm{s} \in A$" avec $A \subseteq \{\texttt{invalid}, \texttt{valid}, \texttt{comm}\}$.
Ici : égalité syntaxique sur les variables (i.e. absence d'alias).

Propriété 1 : $\mathcal{L}_{\mathrm{s}}$ décidable.

Propriété 2 : les méta-invariants $\mathcal{MI}_2$ et $\mathcal{MI}_3$ s'écrivent dans $\mathcal{L}_{\mathrm{s}}$.

Annotations décidables sur les statuts

Logique $\mathcal{L}_{\mathrm{s}}$: logique propositionnelle classique avec atomes de la forme "$x.\mathrm{s} \in A$" avec $A \subseteq \{\texttt{invalid}, \texttt{valid}, \texttt{comm}\}$.
Ici : égalité syntaxique sur les variables (i.e. absence d'alias).

Propriété 1 : $\mathcal{L}_{\mathrm{s}}$ décidable.

Propriété 2 : les méta-invariants $\mathcal{MI}_2$ et $\mathcal{MI}_3$ s'écrivent dans $\mathcal{L}_{\mathrm{s}}$.

On se limite à des annotations sur les statuts dans $\mathcal{L}_{\mathrm{s}}$.

Bref aperçu

Principe interprétation abstraite du calcul de wp (qui permet de rester dans $\mathcal{L}_{\mathrm{S}}$).

Bref aperçu

Principe interprétation abstraite du calcul de wp (qui permet de rester dans $\mathcal{L}_{\mathrm{s}}$).

Types grosso-modo de la forme "P ! $\mathbb{A}$" avec $P \in \mathcal{L}_{\mathrm{s}}$ et

$$\begin{array}{lll} \mathbb{A} ::= & \mid \mathrm{skip} & \\ & \mid x.\mathrm{s}{:=}c \parallel \mathbb{A} & \text{avec } x \text{ nom de var. nouveau} \\ & & \text{et } c \in \{\texttt{invalid}, \texttt{valid}, \texttt{comm}\} \end{array}$$

Bref aperçu

Principe interprétation abstraite du calcul de wp (qui permet de rester dans $\mathcal{L}_{\mathrm{s}}$).

Types grosso-modo de la forme "P ! $\mathbb{A}$" avec $P \in \mathcal{L}_{\mathrm{s}}$ et

$$\begin{array}{rll} \mathbb{A} ::= & \mid \mathrm{skip} & \\ & \mid x.\mathrm{s}{:=}c \parallel \mathbb{A} & \text{avec } x \text{ nom de var. nouveau} \\ & & \text{et } c \in \{\texttt{invalid}, \texttt{valid}, \texttt{comm}\} \end{array}$$

Ppté Si $P, Q \in \mathcal{L}_{\mathrm{s}}$, $wp(P \mid \mathbb{A}, Q)$ se réduit à une proposition de $\mathcal{L}_{\mathrm{s}}$

Bref aperçu

Principe interprétation abstraite du calcul de wp (qui permet de rester dans $\mathcal{L}_{\mathrm{s}}$).

Types grosso-modo de la forme "P ! $\mathbb{A}$" avec $P \in \mathcal{L}_{\mathrm{s}}$ et

$$\begin{array}{lll} \mathbb{A} ::= & \mid \mathrm{skip} & \\ & \mid x.\mathrm{s}{:=}c \parallel \mathbb{A} & \text{avec } x \text{ nom de var. nouveau} \\ & & \text{et } c \in \{\texttt{invalid}, \texttt{valid}, \texttt{comm}\} \end{array}$$

Ppté Si $P, Q \in \mathcal{L}_{\mathrm{s}}$, $wp(P \mid \mathbb{A}, Q)$ se réduit à une proposition de $\mathcal{L}_{\mathrm{s}}$

Egalité de type entre "P_1 ! $\mathbb{A}_1$" et "P_2 ! $\mathbb{A}_2$" se ramène à l'égalité des substitutions :
"$(\mathcal{MI}_2 \wedge \mathcal{MI}_3) \wedge P_1 \mid \mathbb{A}_1$" et "$(\mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge P_2) \mid \mathbb{A}_2$".

Bref aperçu

Principe interprétation abstraite du calcul de wp (qui permet de rester dans $\mathcal{L}_s$).

Types grosso-modo de la forme "P ! $\mathbb{A}$" avec $P \in \mathcal{L}_s$ et

$\mathbb{A}$ ::= | skip
| x.s:=c || $\mathbb{A}$ avec x nom de var. nouveau et $c \in \{\texttt{invalid}, \texttt{valid}, \texttt{comm}\}$

Ppté Si $P, Q \in \mathcal{L}_s$, $wp(P \mid \mathbb{A}, Q)$ se réduit à une proposition de $\mathcal{L}_s$

Egalité de type entre "P_1 ! $\mathbb{A}_1$" et "P_2 ! $\mathbb{A}_2$" se ramène à l'égalité des substitutions :
"$(\mathcal{MI}_2 \wedge \mathcal{MI}_3) \wedge P_1 \mid \mathbb{A}_1$" et "$(\mathcal{MI}_2 \wedge \mathcal{MI}_3 \wedge P_2) \mid \mathbb{A}_2$".

Exemples où x owns z et y owns z.

```
unpack(x); unpack(z); z:=!z+1; pack(z); pack(x)
    (* a le type ''x.s ∈ {valid} ! skip'' *)
unpack(x); unpack(y)   (* pré. insatisfiable => rejeté *)
unpack(x); z:=!z+1     (* idem *)
```

Plan

Perspectives

- Extension de B avec raffinement.
- Adaptation à WHY (avec implémentation) :
  - langage d'expressions (versus instruction en B).
  - invariants sur états locaux (plusieurs instances d'état par "modules").
  - validation par traduction fonctionnelle.
- Généralisation avec des alias ?