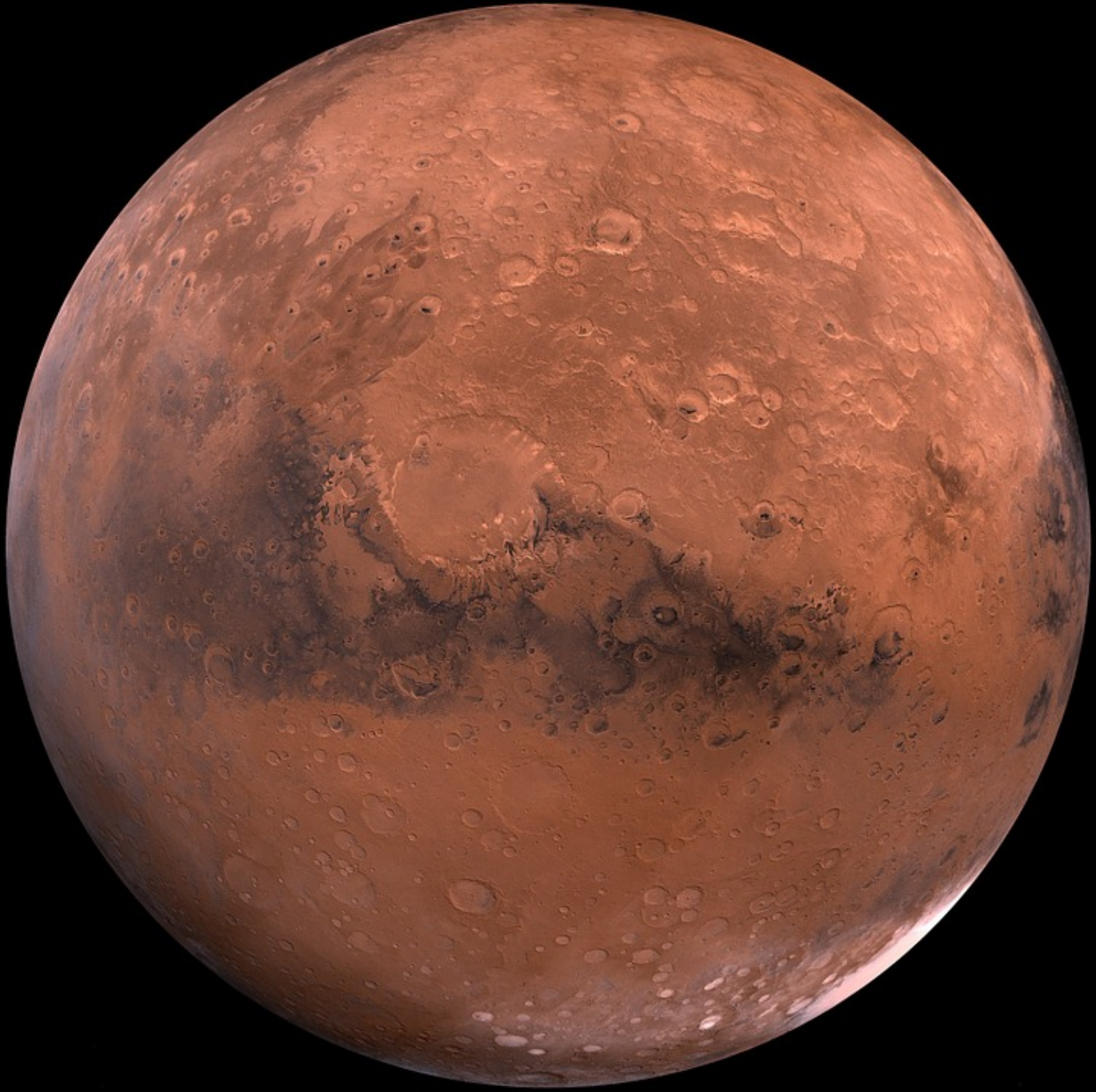
Vérification par interprétation abstraite en mémoire faiblement cohérente

Thibault Suzanne
Thèse dirigée par Antoine Miné

26 février 2019

École normale supérieure



Pathfinder

Pathfinder

Perte d'information à cause d'un bug de *parallélisme* !

Qu’est-ce qu’un programme ?

```
let explore abs constrs =
  let rec aux abs cstrs res depth =
    match consistency abs cstrs with
    | Empty -> res
    | Full abs -> add_s res abs
    | Maybe (abs, cstrs, _) ->
      if Abs.is_small abs || depth >= !Constant.max_iter then add_u res abs
      else List.fold_left (fun res elem -> aux elem cstrs (incr_step res) (depth + 1))
          res (split abs cstrs)
  in aux abs constrs empty_res 0
```

- Un ensemble d’instructions exécutées par un ordinateur.
- L’ordinateur fait ce qu’on lui dit, pas ce à quoi on pense !
- Quand les deux ne correspondent pas, on observe un bug.

Qu’est-ce qu’un programme ?

```
let explore abs constrs =
  let rec aux abs cstrs res depth =
    match consistency abs cstrs with
    | Empty -> res
    | Full abs -> add_s res abs
    | Maybe (abs, cstrs, _) ->
      if Abs.is_small abs || depth >= !Constant.max_iter then add_u res abs
      else List.fold_left (fun res elem -> aux elem cstrs (incr_step res) (depth + 1))
          res (split abs cstrs)
  in aux abs constrs empty_res 0
```

- Un ensemble d’instructions exécutées par un ordinateur.
- L’ordinateur fait ce qu’on lui dit, pas ce à quoi on pense !
- Quand les deux ne correspondent pas, on observe un bug.

Qu’est-ce qu’un programme ?

```
let explore abs constrs =
  let rec aux abs cstrs res depth =
    match consistency abs cstrs with
    | Empty -> res
    | Full abs -> add_s res abs
    | Maybe (abs, cstrs, _) ->
      if Abs.is_small abs || depth >= !Constant.max_iter then add_u res abs
      else List.fold_left (fun res elem -> aux elem cstrs (incr_step res) (depth + 1))
          res (split abs cstrs)
  in aux abs constrs empty_res 0
```

- Un ensemble d’instructions exécutées par un ordinateur.
- L’ordinateur fait ce qu’on lui dit, pas ce à quoi on pense !
- Quand les deux ne correspondent pas, on observe un bug.

Parallélisme et cohérence faible

L'amélioration de puissance des processeurs modernes se traduit par la multiplication des cœurs.

$\Longrightarrow$ Pour en profiter : programmation parallèle

```
x = 1;  | y = 1;
r0 = y; | r1 = x;
```

L'intuition du programmeur : la cohérence séquentielle

En fin d'exécution, `r0 = 1 || r1 = 1`.

Sur un processeur x86...

On observe parfois `r0 = 0 && r1 = 0` !

Parallélisme et cohérence faible

L'amélioration de puissance des processeurs modernes se traduit par la multiplication des cœurs.

$\Longrightarrow$ Pour en profiter : programmation parallèle

```
x = 1;  | y = 1;
r0 = y; | r1 = x;
```

L'intuition du programmeur : la cohérence séquentielle
En fin d'exécution, `r0 = 1 || r1 = 1`.

Sur un processeur x86...
On observe parfois `r0 = 0 && r1 = 0` !

Parallélisme et cohérence faible

L'amélioration de puissance des processeurs modernes se traduit par la multiplication des cœurs.

$\Longrightarrow$ Pour en profiter : programmation parallèle

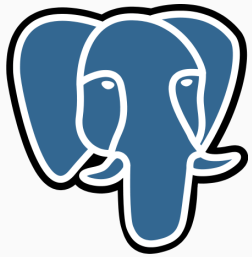
```
x = 1;  | y = 1;
r0 = y; | r1 = x;
```

L'intuition du programmeur : la cohérence séquentielle
En fin d'exécution, `r0 = 1 || r1 = 1`.

Sur un processeur x86...
On observe parfois `r0 = 0 && r1 = 0` !

PostgreSQL

```
#define WORKERS 2
volatile _Bool latch [WORKERS];
volatile _Bool flag [WORKERS];
void worker(int i) {
    while (!latch[i]);
    for (;;) {
        latch[i] = 0;

        if (flag[i]) {
            flag[i] = 0;
            flag[(i + 1) % WORKERS] = 1;

            latch[(i + 1) % WORKERS] = 1;
        }
        while (!latch[i]);
    }
}
```



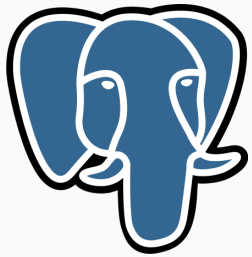
- Passage de jeton dans PostgreSQL
- Fonctionne correctement sur x86
- PowerPC : interblocages observés

Bug dû au modèle mémoire !

J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems*, pages 512 – 532. Springer, 2013.

PostgreSQL

```
#define WORKERS 2
volatile _Bool latch [WORKERS];
volatile _Bool flag [WORKERS];
void worker(int i) {
    while (!latch[i]);
    for (;;) {
        latch[i] = 0;
        __lwsync();
        if (flag[i]) {
            flag[i] = 0;
            flag[(i + 1) % WORKERS] = 1;
            __lwsync();
            latch[(i + 1) % WORKERS] = 1;
        }
        while (!latch[i]);
    }
}
```



- Passage de jeton dans PostgreSQL
- Fonctionne correctement sur x86
- PowerPC : interblocages observés

Bug dû au modèle mémoire !

J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems*, pages 512 – 532. Springer, 2013.

Vérification automatique

- Il est difficile de s'assurer qu'un programme est correct
- Parallélisme et cohérence faible compliquent encore la tâche

Il devient alors nécessaire de disposer d'outils de vérification *automatiques* : des programmes qui analysent d'autres programmes.

Objectif de cette thèse

- Vérifier automatiquement des programmes dans un environnement faiblement cohérent
- Analyse *sûre* qui n'oublie aucun comportement possible
- Dans le cadre théorique de l'*interprétation abstraite*
- Programmes numériques en quasi-assembleur

Plan

1. Contexte
2. Analyse monolithique
3. Analyse modulaire
4. Abstraction avancées
5. Conclusion

Contexte

Contexte

Interprétation abstraite

Objectif

```
(ℓ1) x = [0, 100];

(ℓ2) while (ℓ3) (x < 1000) {
       (ℓ4) x = x + 10 * [1, 10]; (ℓ5)
     } (ℓ6)
```

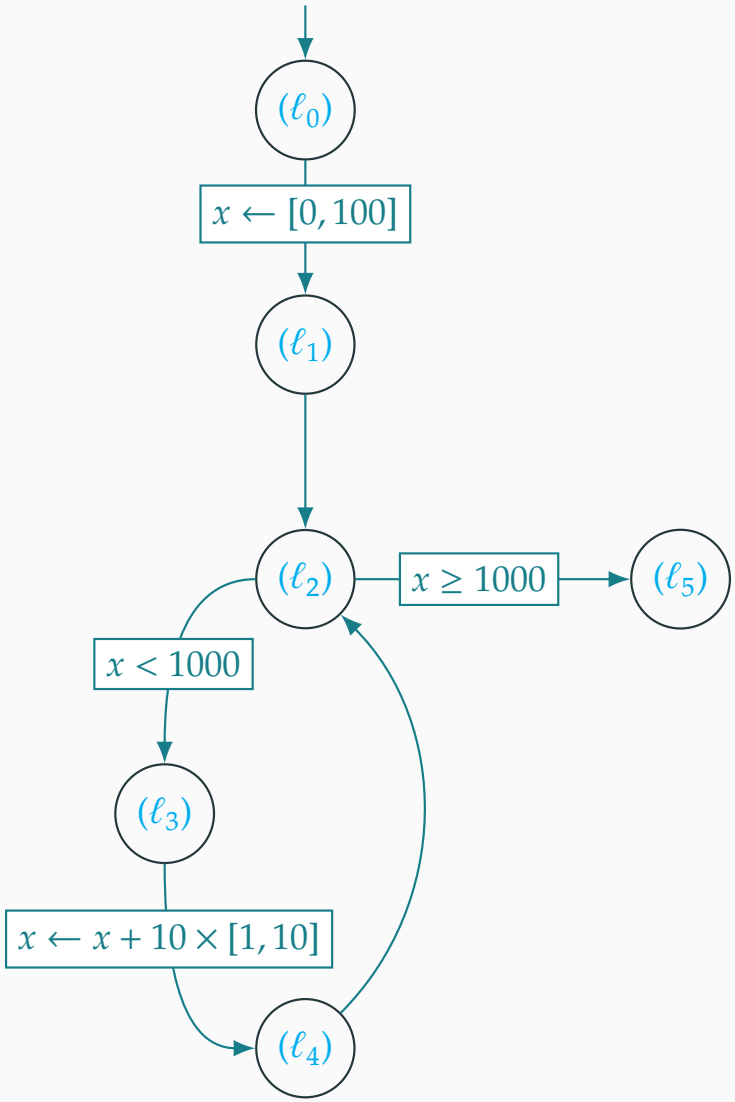
- On cherche à calculer toutes les exécutions possibles
- Indécidable dans le cas général : on doit *abstraire* pour perdre une précision contrôlée

Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
       (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
} (ℓ5)
```

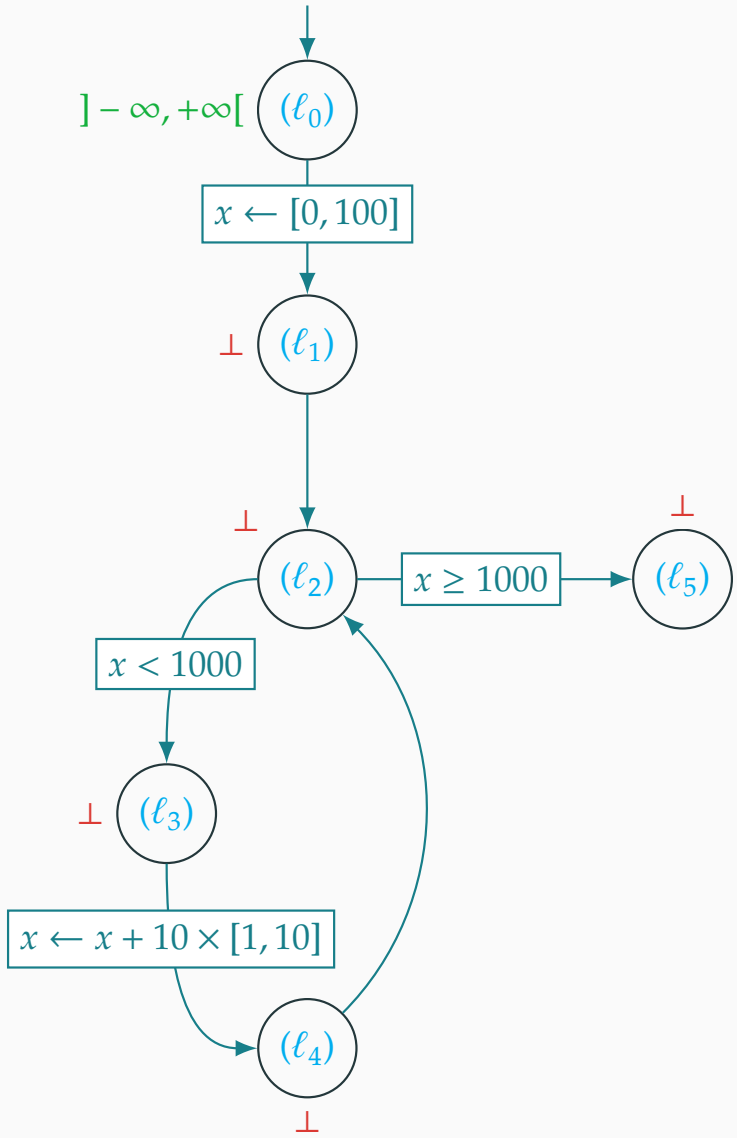
Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
       (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
}    (ℓ5)
```



Exemple : les intervalles

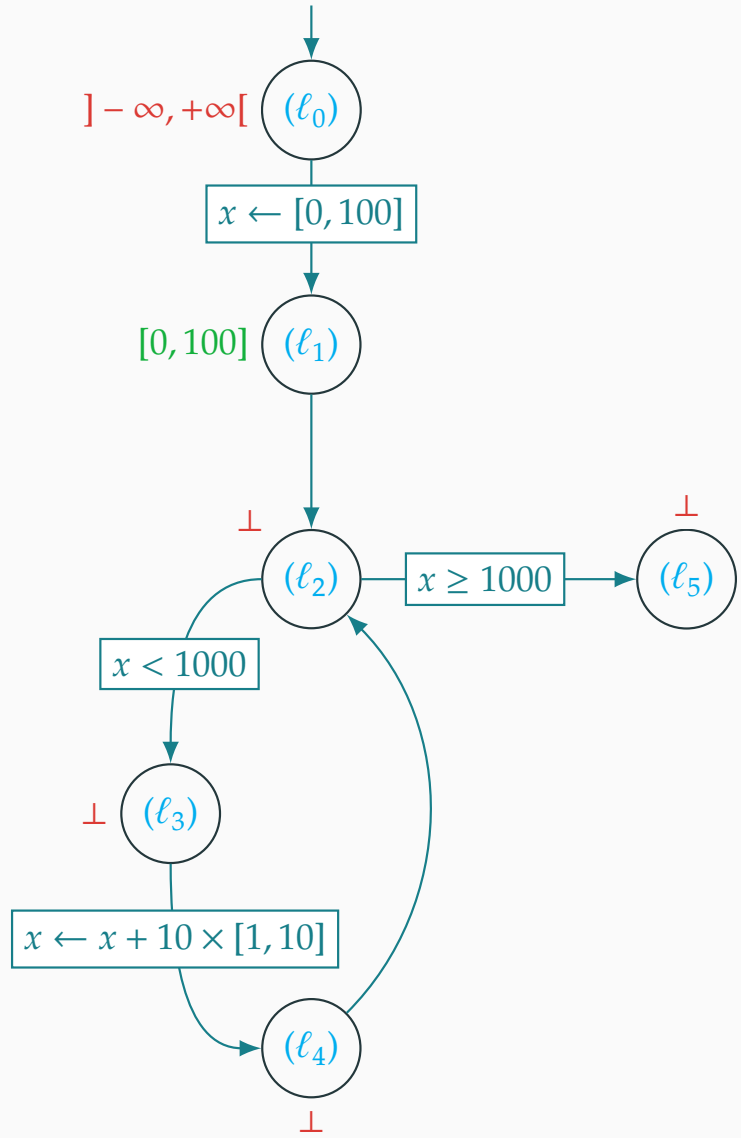
```
(ℓ0)  x = [0, 100];

(ℓ1)  while (ℓ2) (x < 1000) {
        (ℓ3)  x = x + 10 * [1, 10];  (ℓ4)
      }  (ℓ5)
```



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
       (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
}    (ℓ5)
```

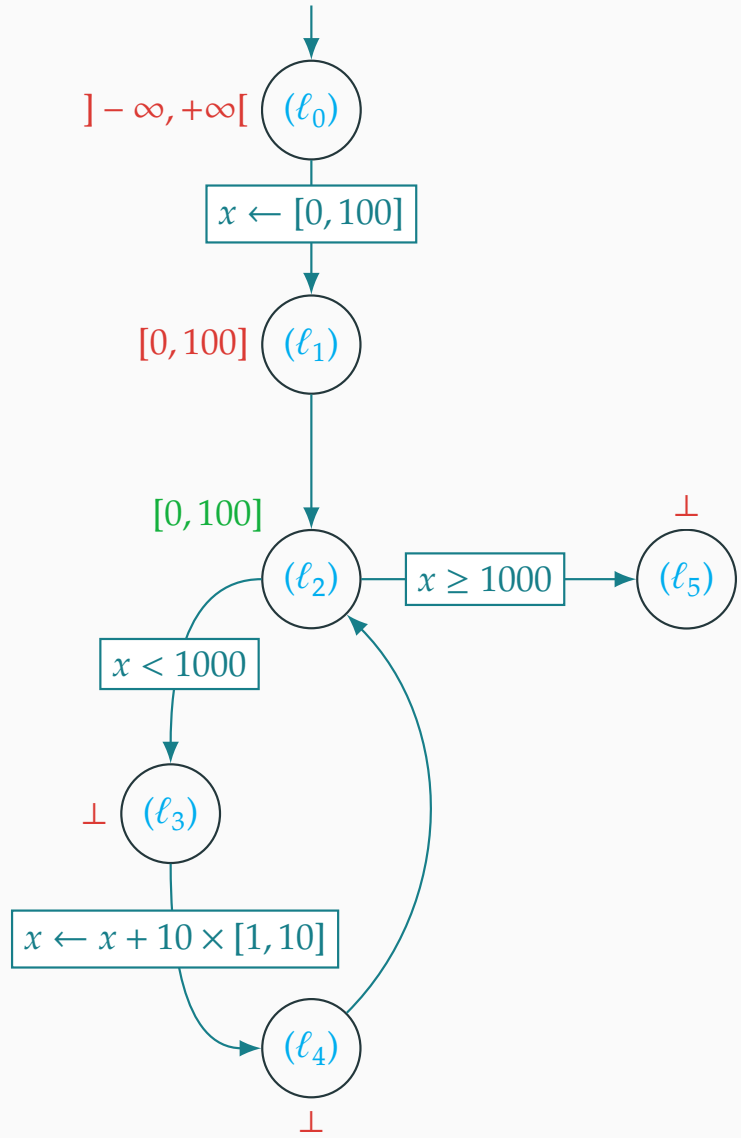
- Exécution dans l'abstrait



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
       (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
}    (ℓ5)
```

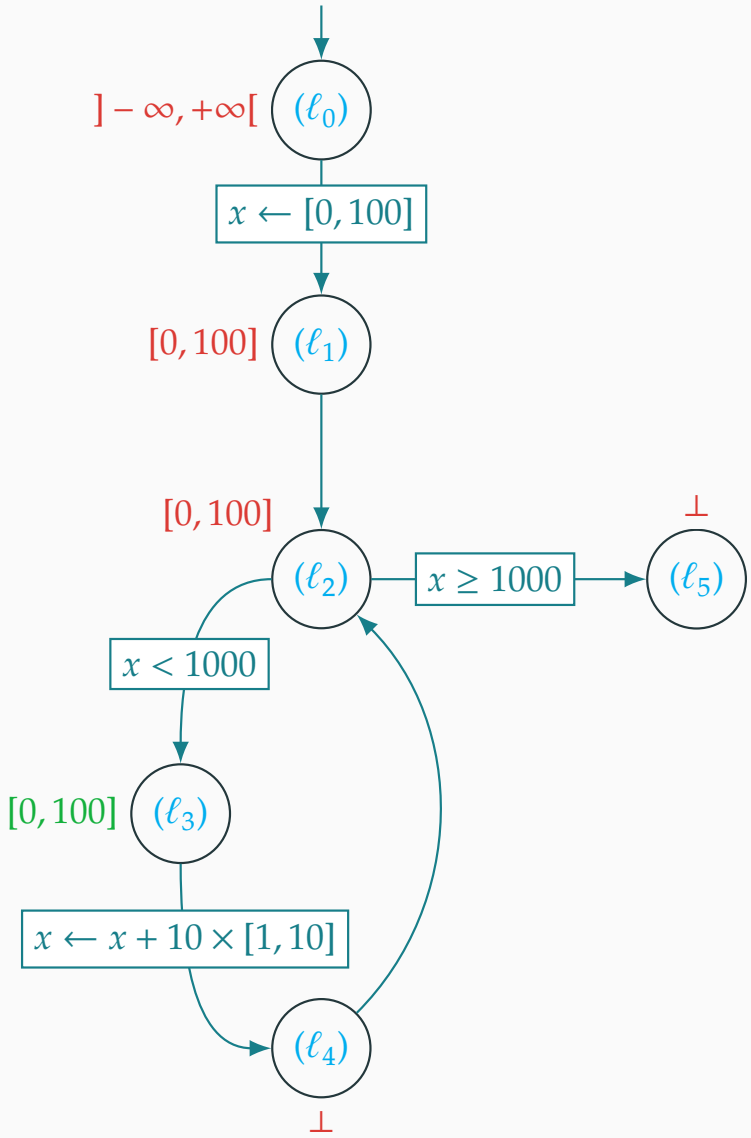
- Exécution dans l'abstrait



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
     (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
} (ℓ5)
```

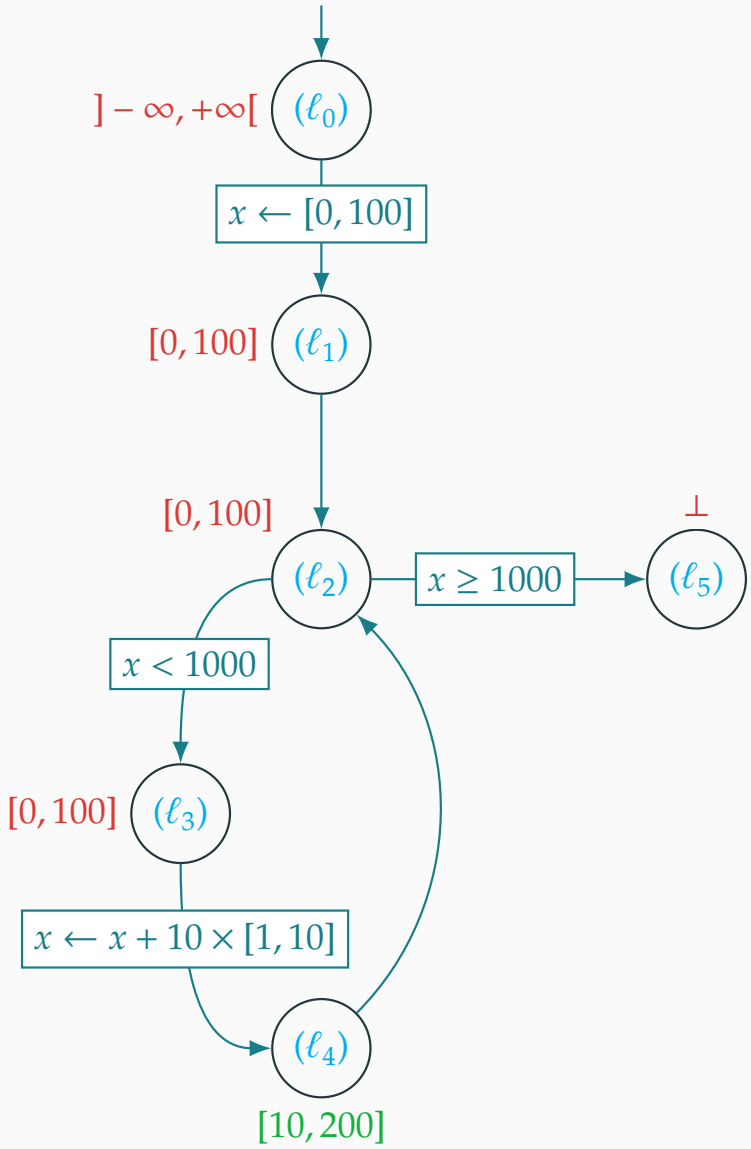
- Exécution dans l'abstrait



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
     (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
} (ℓ5)
```

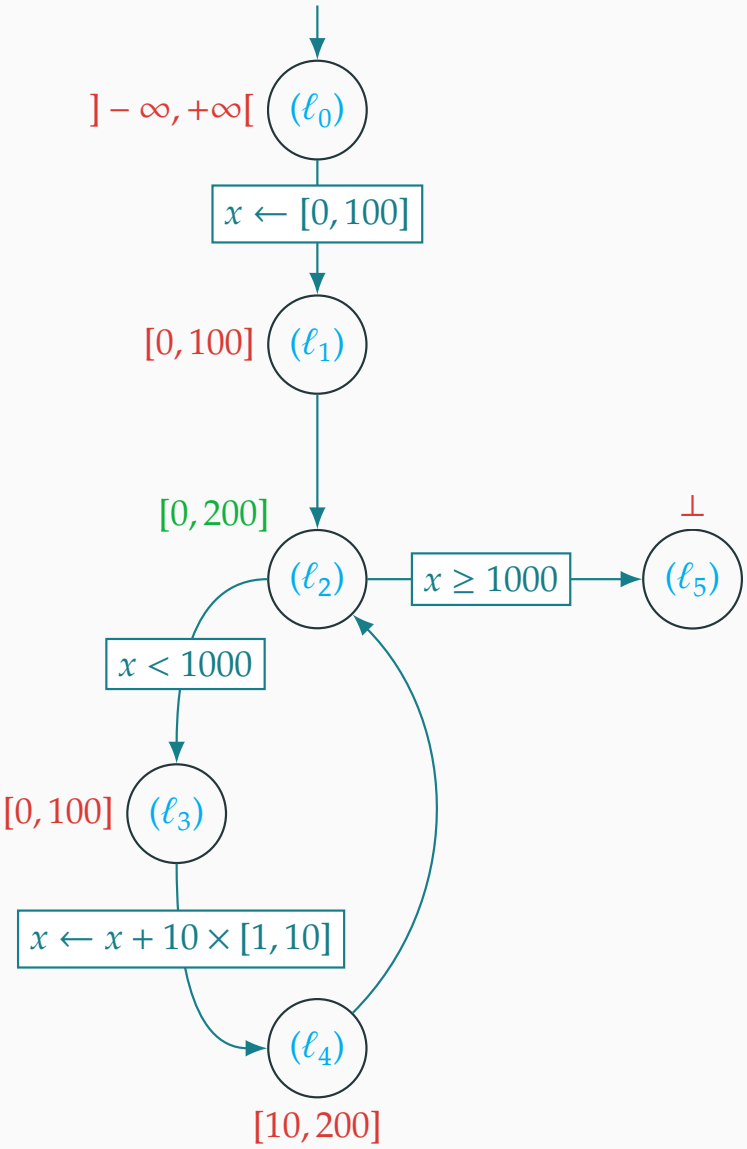
- Exécution dans l'abstrait



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
     (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
} (ℓ5)
```

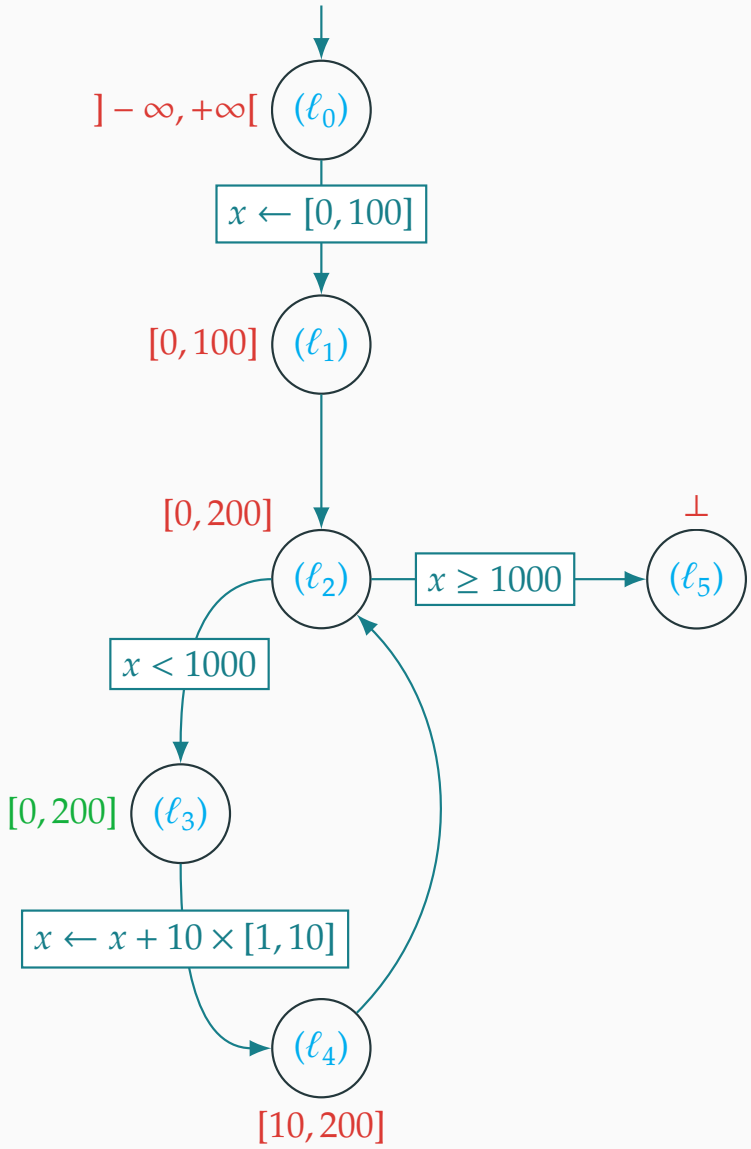
- Exécution dans l'abstrait
- Accumulation des résultats



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
       (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
}    (ℓ5)
```

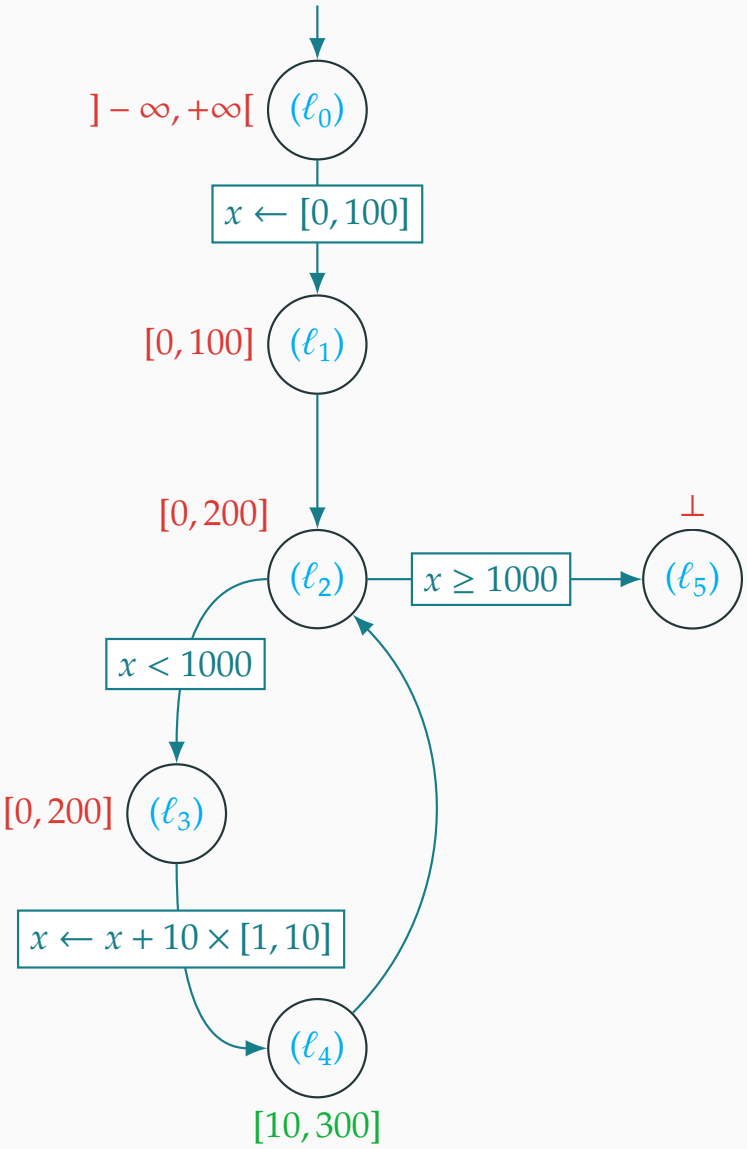
- Exécution dans l'abstrait
- Accumulation des résultats



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
     (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
} (ℓ5)
```

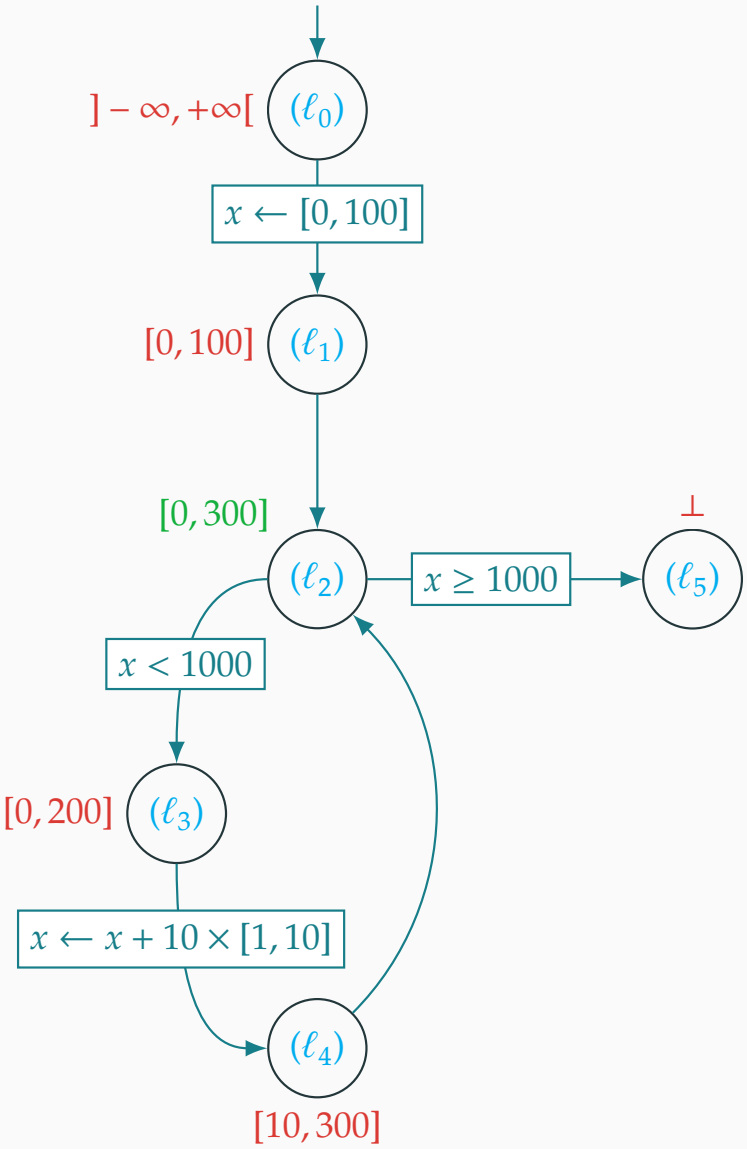
- Exécution dans l'abstrait
- Accumulation des résultats



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
     (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
} (ℓ5)
```

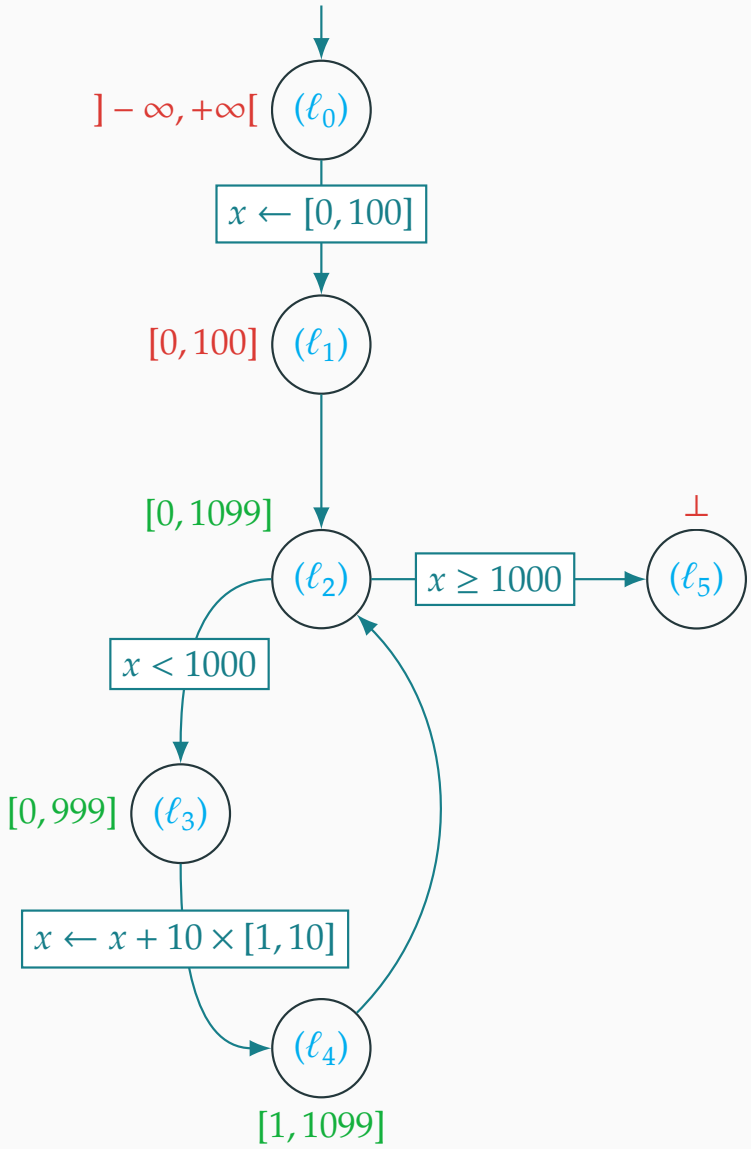
- Exécution dans l'abstrait
- Accumulation des résultats



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
     (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
} (ℓ5)
```

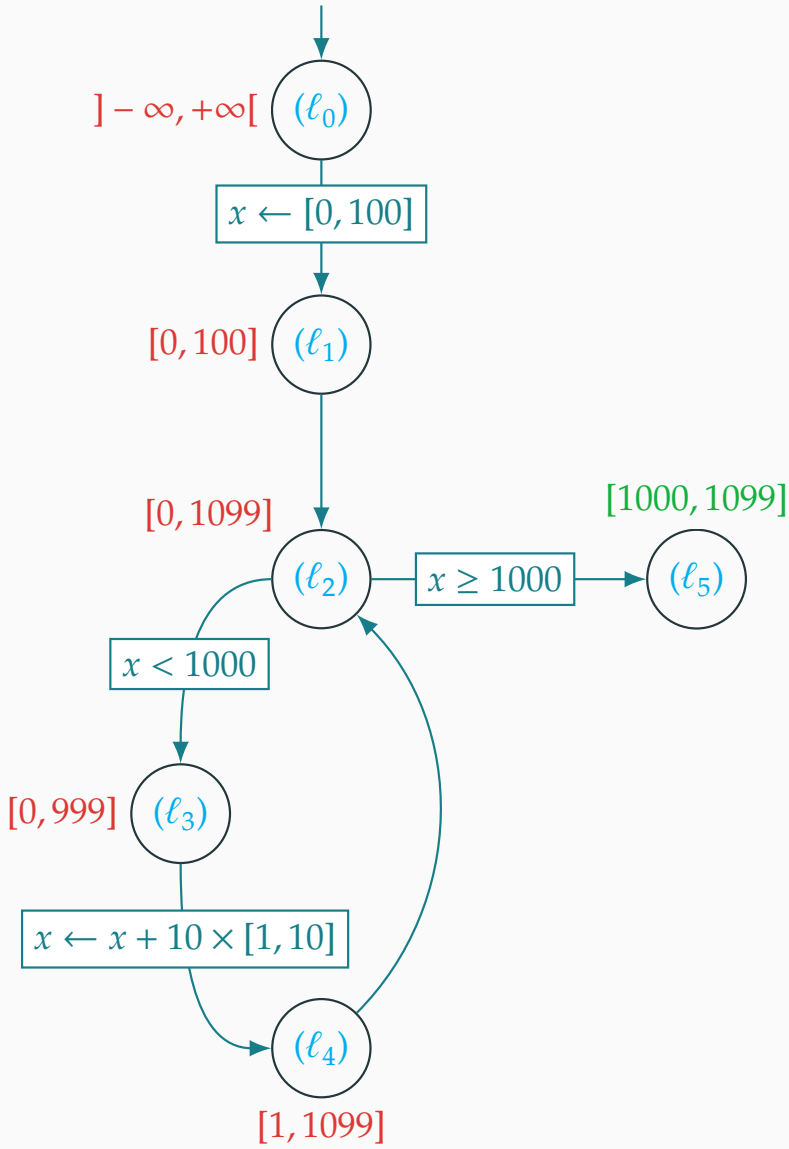
- Exécution dans l'abstrait
- Accumulation des résultats
- Itération vers un point fixe



Exemple : les intervalles

```
(ℓ0) x = [0, 100];

(ℓ1) while (ℓ2) (x < 1000) {
     (ℓ3) x = x + 10 * [1, 10]; (ℓ4)
} (ℓ5)
```

- Exécution dans l'abstrait
- Accumulation des résultats
- Itération vers un point fixe



Formalisation

Par concrétisation :

$$\gamma : \mathscr{D}^\sharp \to \mathscr{D}$$

$$\gamma([a,b]) = \{a, a+1, \dots, b-1, b\}$$

Par correspondance de Galois :

$$\mathscr{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathscr{D}^\sharp$$

$$\forall X \in \mathscr{D}, X^\sharp \in \mathscr{D}^\sharp, \alpha(X) \sqsubseteq^\sharp X^\sharp \iff X \sqsubseteq \gamma(X^\sharp)$$

$$\alpha(X) = [\min X, \max X]$$

$X \sqsubseteq \gamma(X^\sharp) \implies X^\sharp$ est une abstraction sûre de X

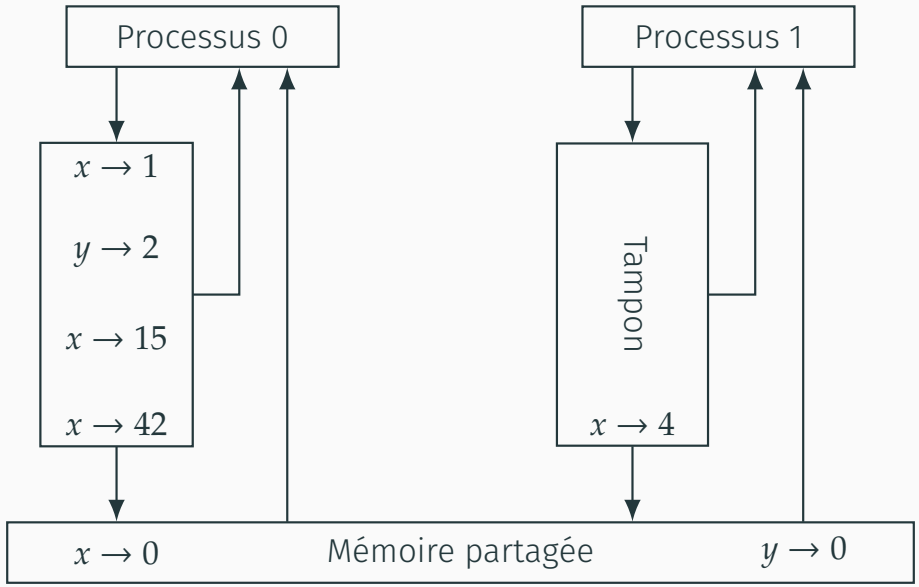
$F : \mathscr{D} \to \mathscr{D}$ a une version abstraite sûre $F^\sharp : \mathscr{D}^\sharp \to \mathscr{D}^\sharp$

Contexte

Mémoire faiblement cohérente

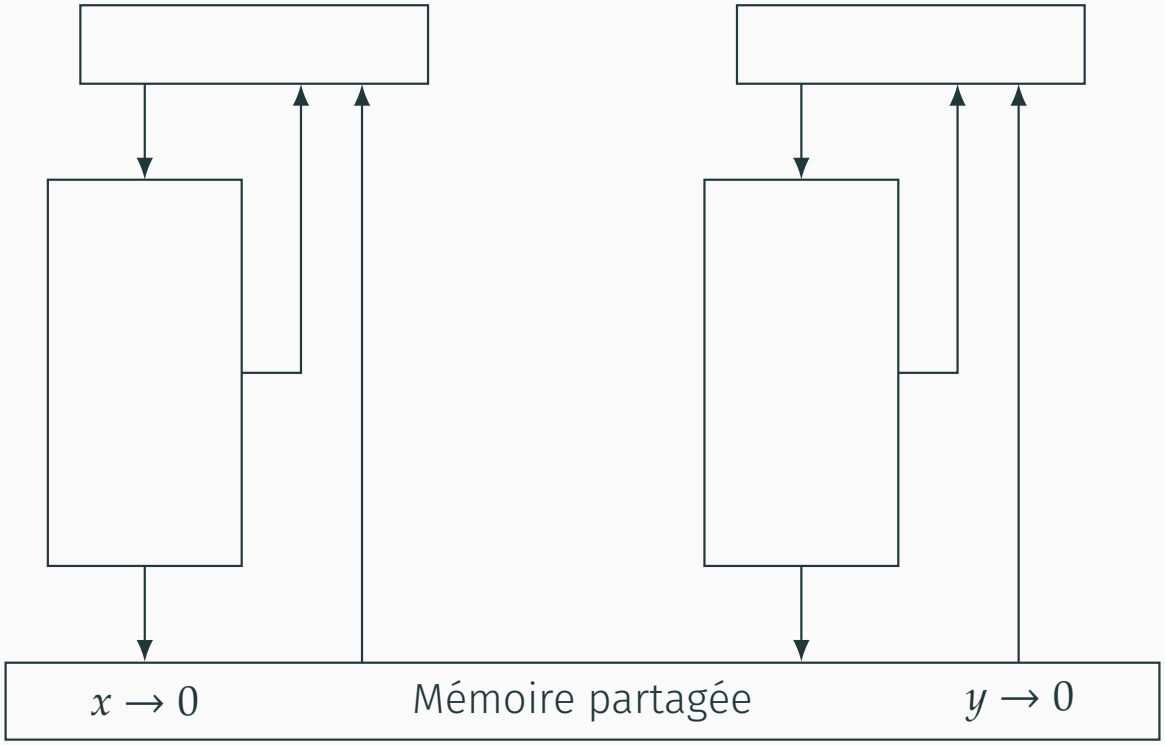
Total Store Ordering : le modèle de x86

- Un tampon par cœur/processus
- Les écritures sont mises en tampon avant la mémoire
- Les lectures se font en priorité depuis le tampon
- Les entrées d'un tampon sont transférées :
  - De façon non déterministe
  - Dans l'ordre d'arrivée (*FIFO*)
- La barrière `mfence` transfère tout le contenu d'un tampon



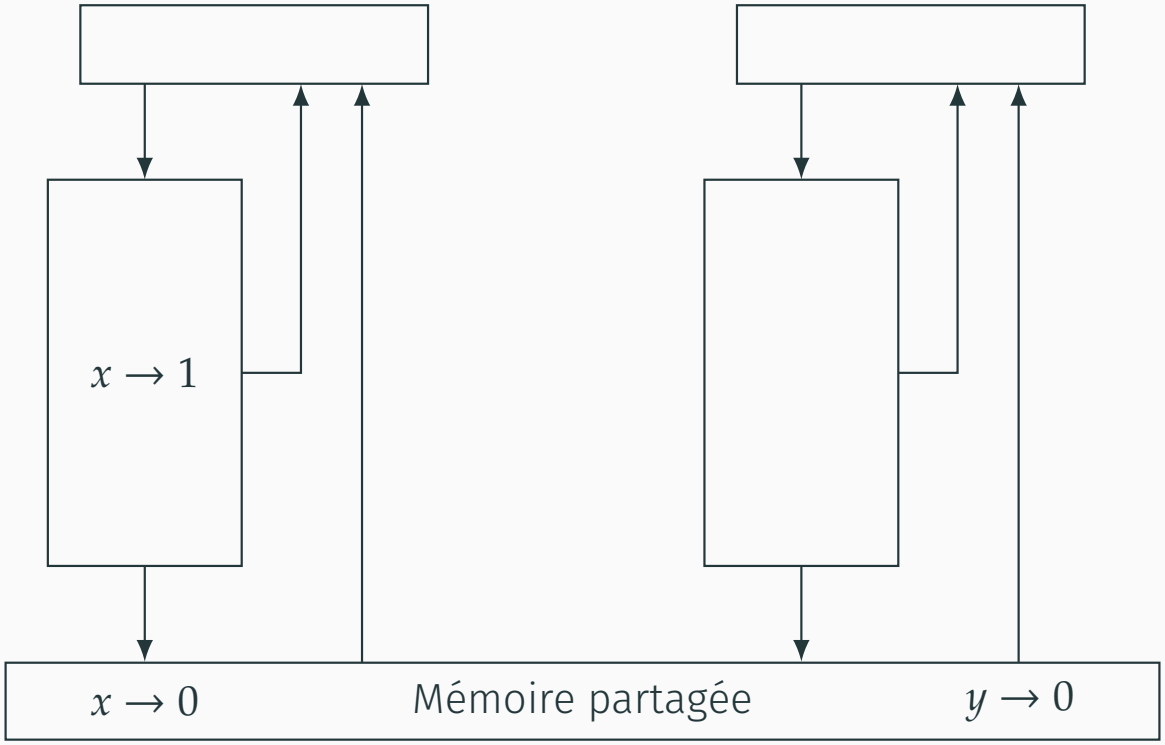
Exemple d'exécution

```
x = 1;  | y = 1;
r0 = y; | r1 = x;
```



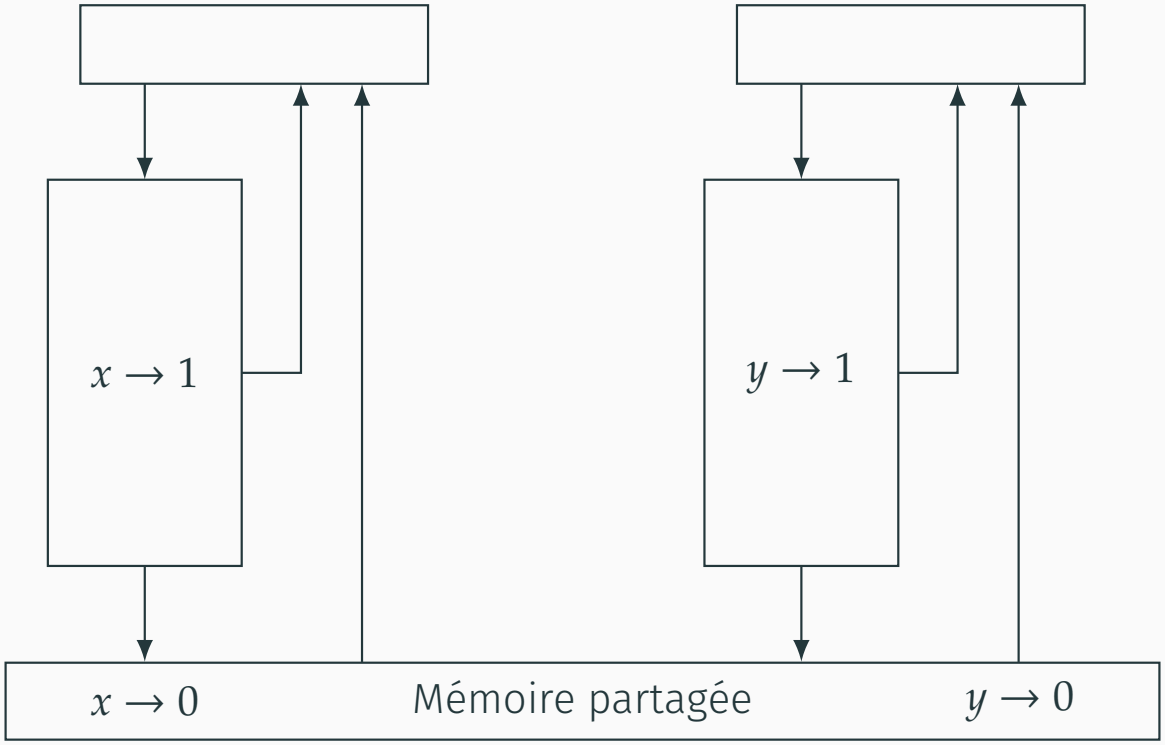
Exemple d'exécution

```
x = 1;  |  y = 1;
r0 = y; |  r1 = x;
```



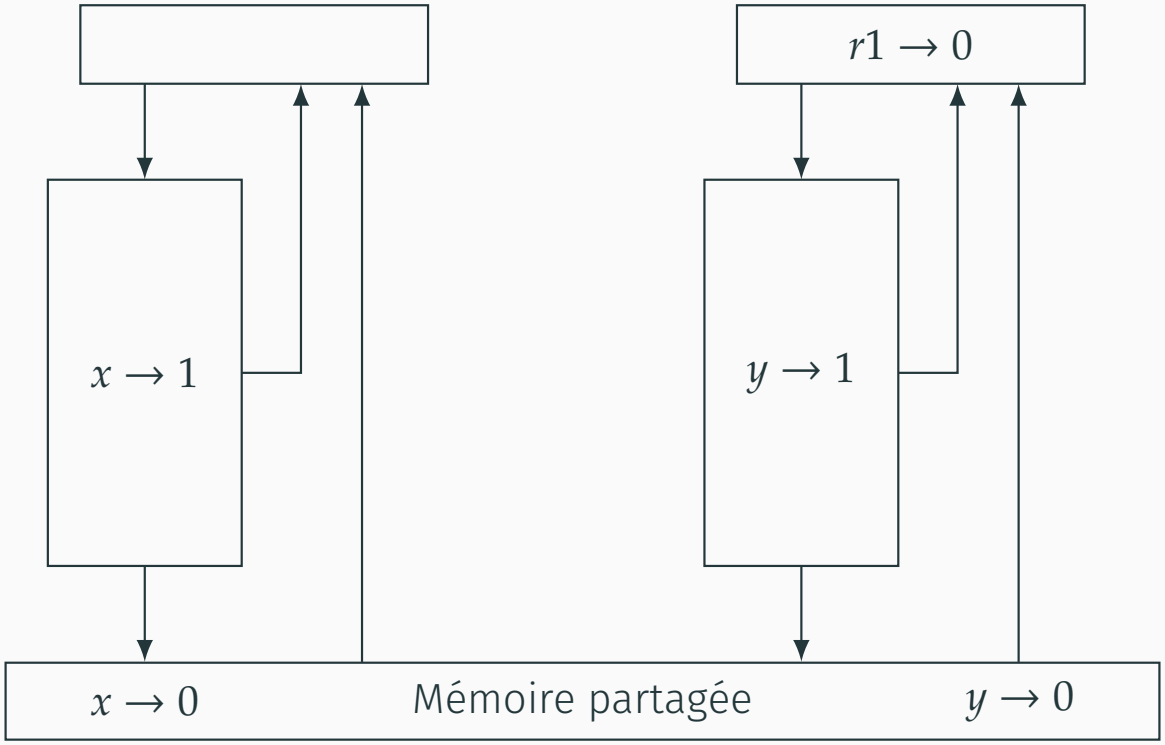
Exemple d'exécution

```
x = 1;  | y = 1;
r0 = y; | r1 = x;
```



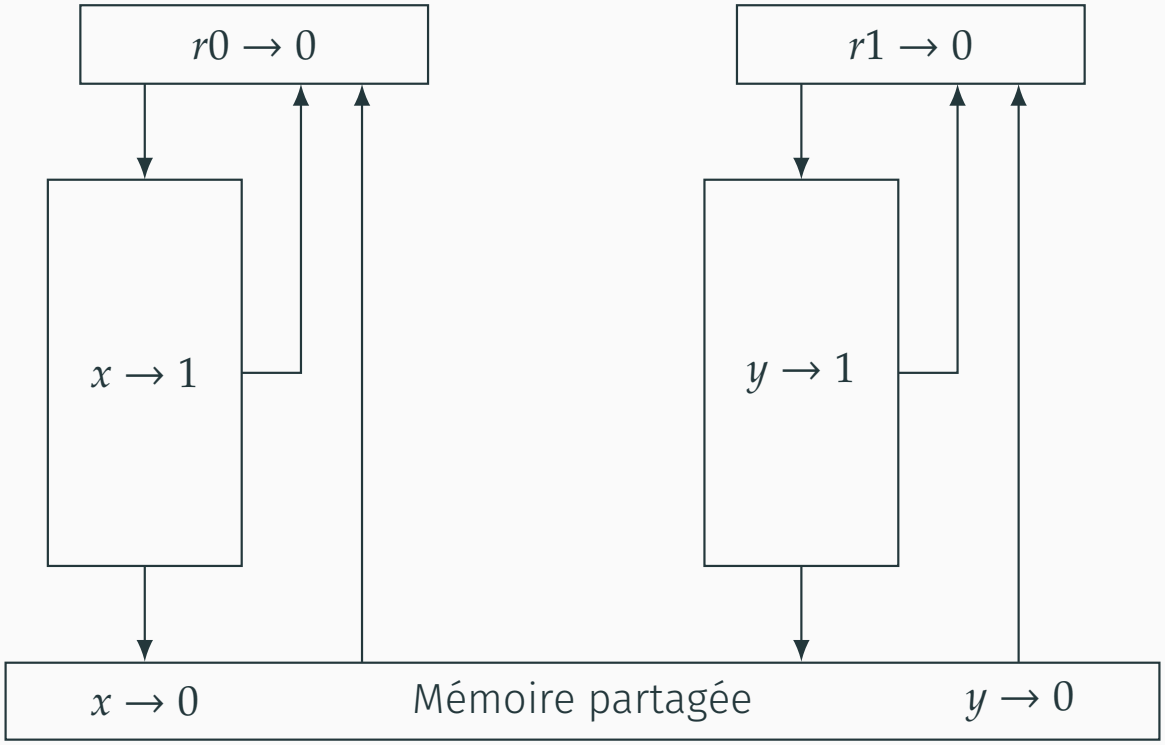
Exemple d'exécution

```
x = 1;  | y = 1;
r0 = y; | r1 = x;
```



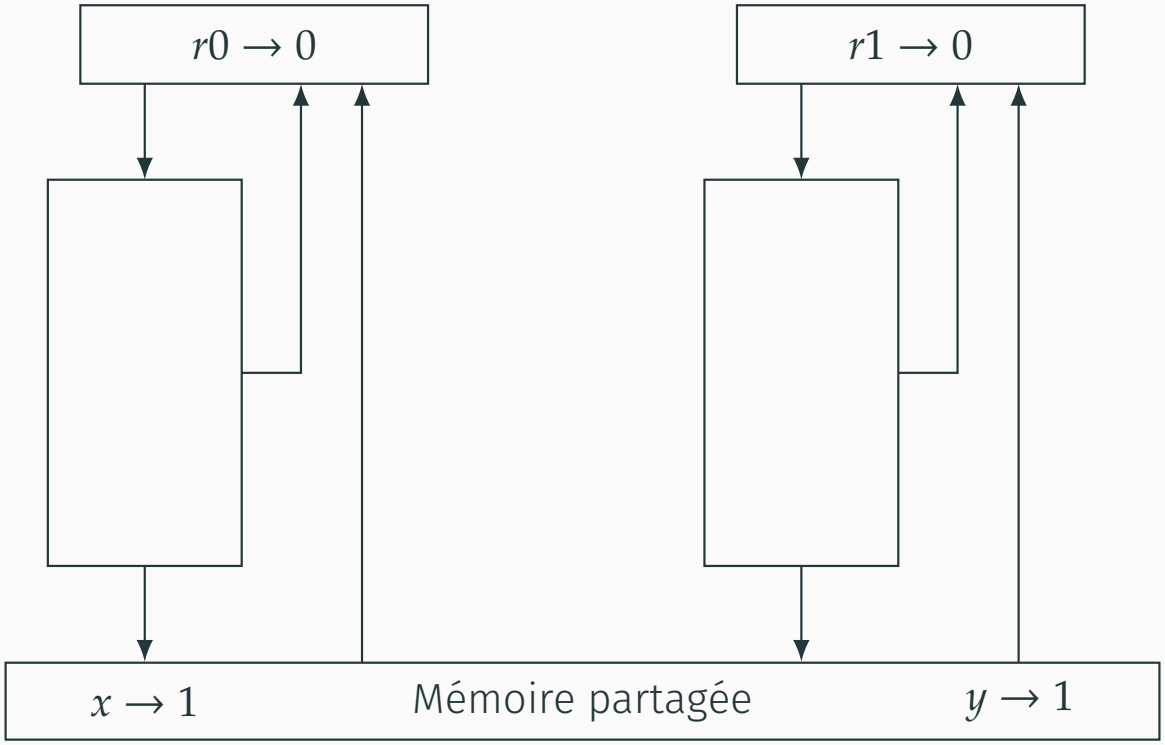
Exemple d'exécution

```
x = 1;  | y = 1;
r0 = y; | r1 = x;
```



Exemple d'exécution

```
x = 1;  | y = 1;
r0 = y; | r1 = x;
```



Analyse monolithique

Graphe des entrelacements

- Représentation des entrelacements par le graphe produit
- Non-déterminisme des transferts : ajout d'une boucle $[\![flush]\!]$ sur chaque nœud
- Puis calcul de point fixe classique
  - Résultat *clos par transfert*
  - Barrières modélisées par un filtre « tampons vides »

Graphe des entrelacements

- Représentation des entrelacements par le graphe produit
- Non-déterminisme des transferts : ajout d'une boucle $[\![flush]\!]$ sur chaque nœud
- Puis calcul de point fixe classique
  - Résultat *clos par transfert*
  - Barrières modélisées par un filtre « tampons vides »

Graphe des entrelacements

- Représentation des entrelacements par le graphe produit
- Non-déterminisme des transferts : ajout d'une boucle $[\![flush]\!]$ sur chaque nœud
- Puis calcul de point fixe classique
  - Résultat *clos par transfert*
  - Barrières modélisées par un filtre « tampons vides »

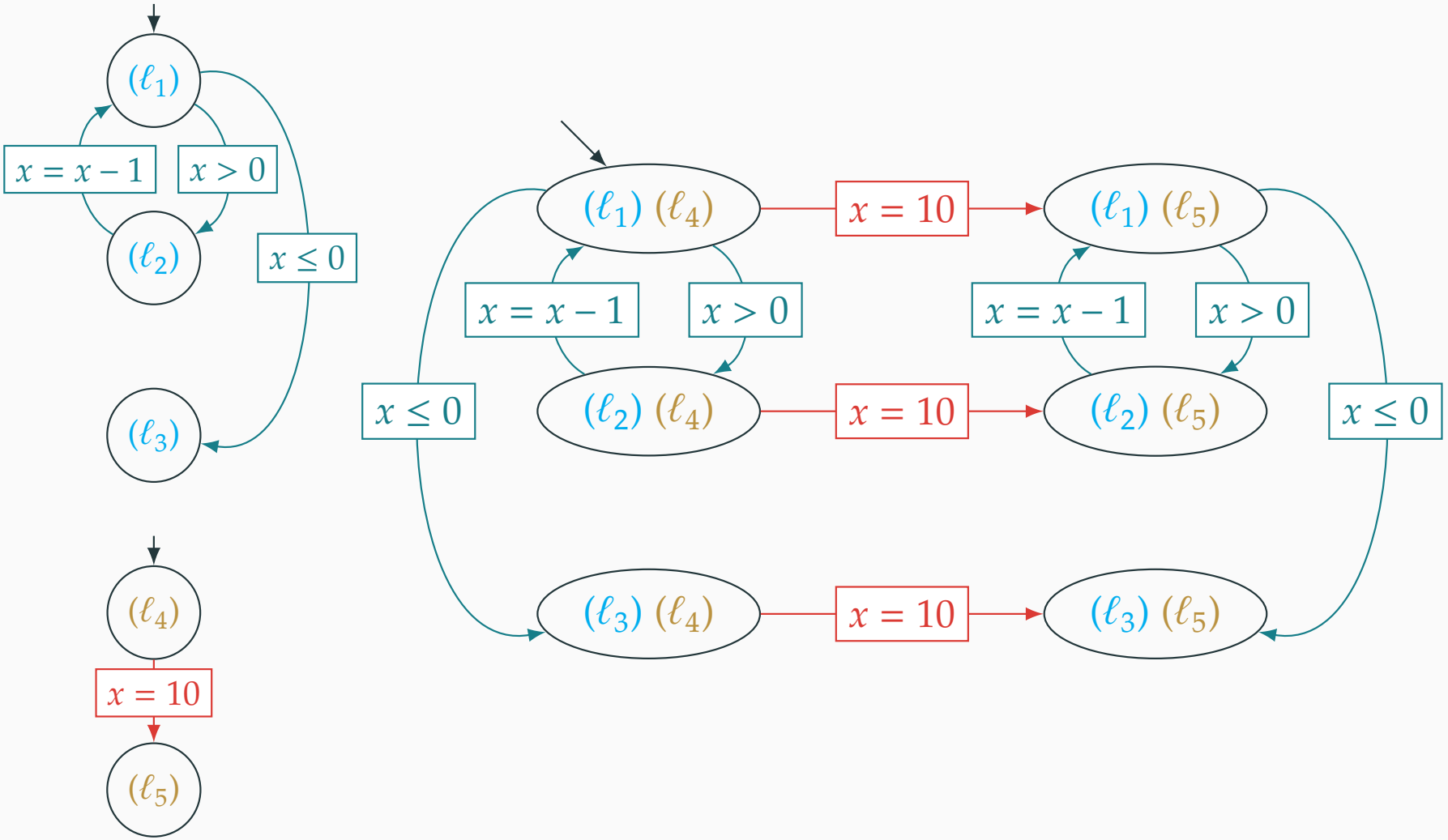
Graphe des entrelacements

- Représentation des entrelacements par le graphe produit
- Non-déterminisme des transferts : ajout d'une boucle $[\![flush]\!]$ sur chaque nœud
- Puis calcul de point fixe classique
  - Résultat *clos par transfert*
  - Barrières modélisées par un filtre « tampons vides »

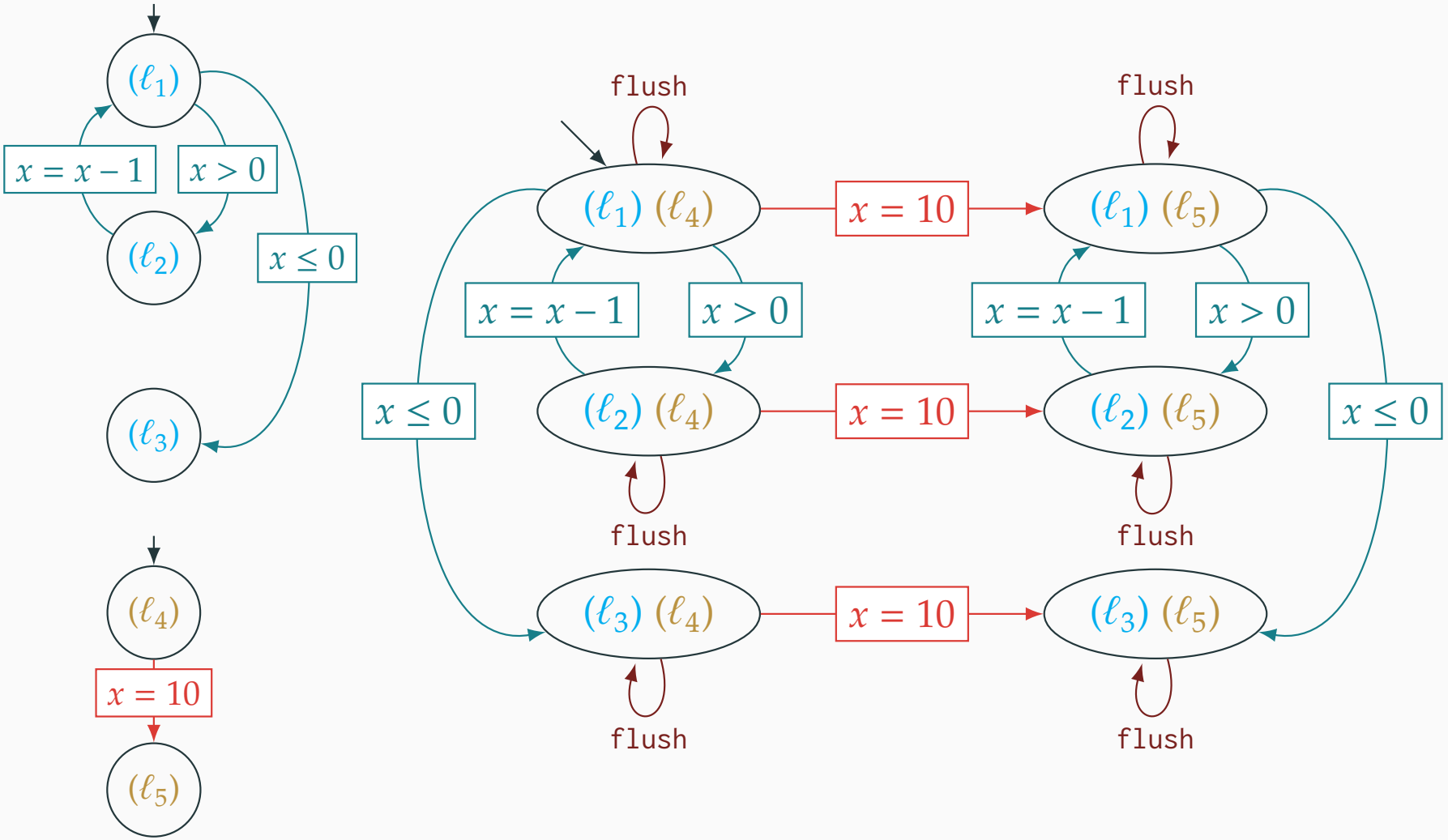
Graphe des entrelacements

- Représentation des entrelacements par le graphe produit
- Non-déterminisme des transferts : ajout d'une boucle $[\![flush]\!]$ sur chaque nœud
- Puis calcul de point fixe classique
  - Résultat *clos par transfert*
  - Barrières modélisées par un filtre « tampons vides »

Graphe produit : exemple



Graphe produit : exemple



Analyse monolithique

Représentation des tampons

Idée générale de l'abstraction

La difficulté vient des tampons, de taille :

- Non bornée
- Mise à jour :
  - Dynamiquement
  - De façon non déterministe

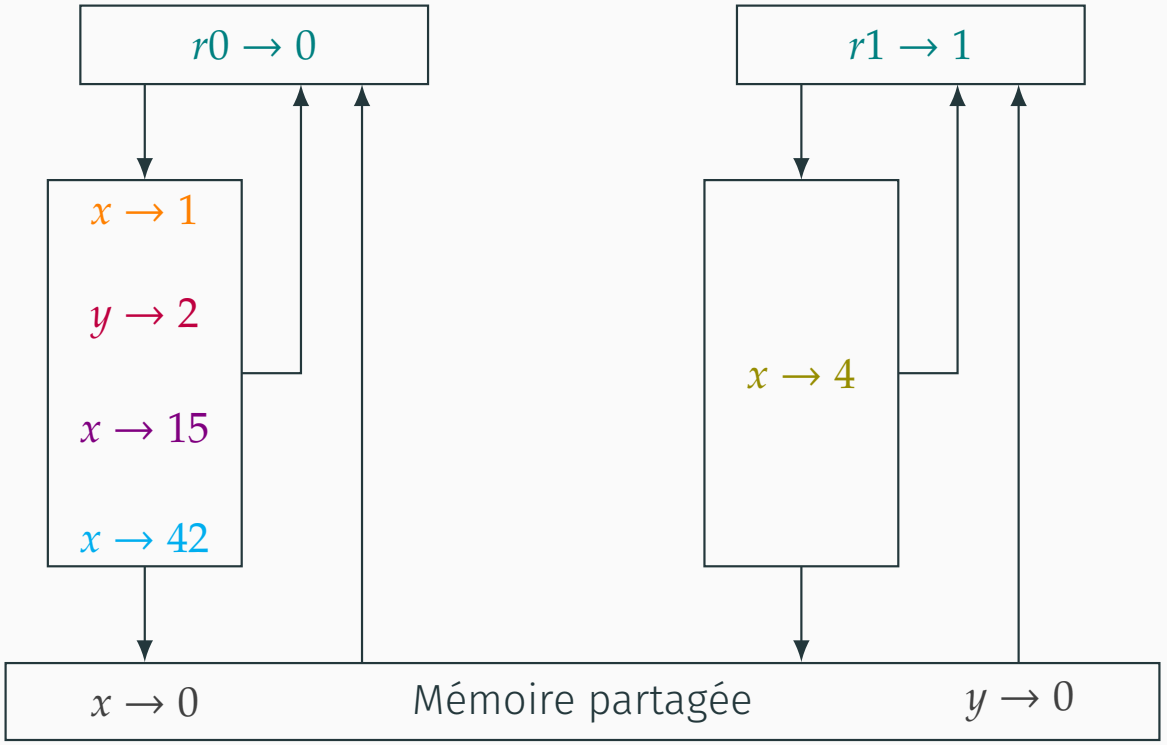
Proposition : adapter des abstractions de tableaux (en particulier la condensation[1])

[1] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *TACAS 2004*.

Encodage par variables — Partial Store Ordering

Une pseudo-variable par entrée du tampon.

L'ordre inter-variables est oublié par l'abstraction.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

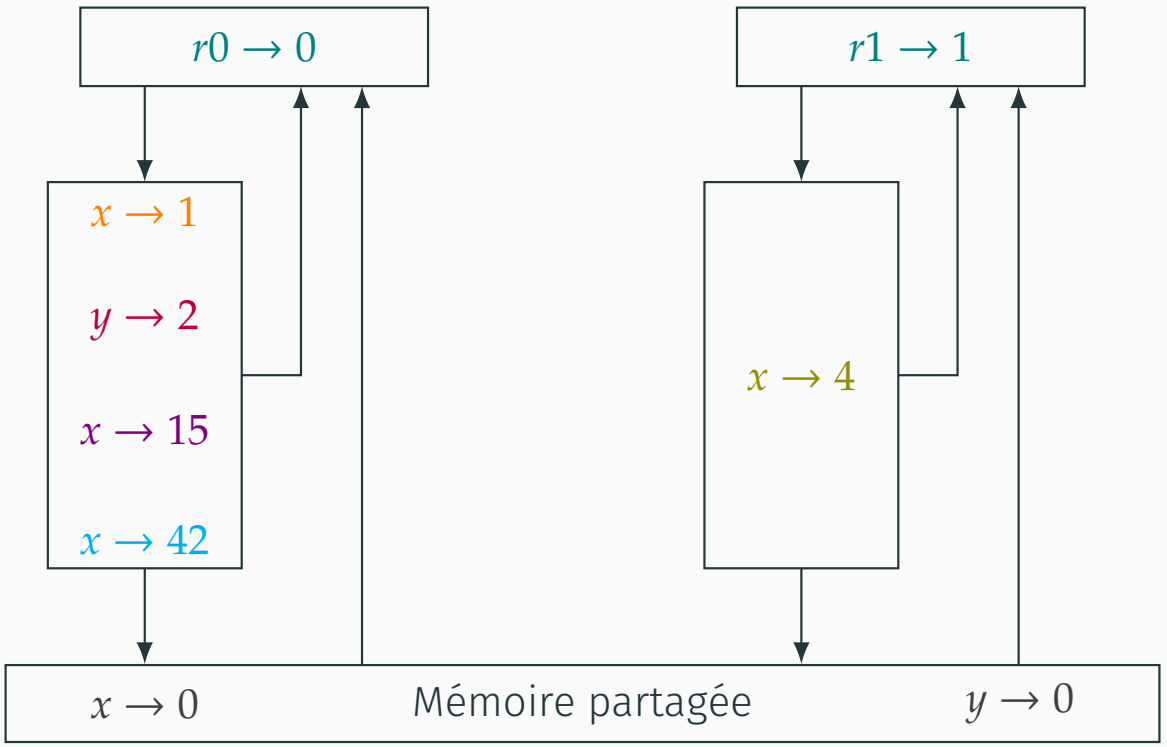
$x^{mem} = 0$

$y^{mem} = 1$

Encodage par variables — Partial Store Ordering

Une pseudo-variable par entrée du tampon.

L'ordre inter-variables est oublié par l'abstraction.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

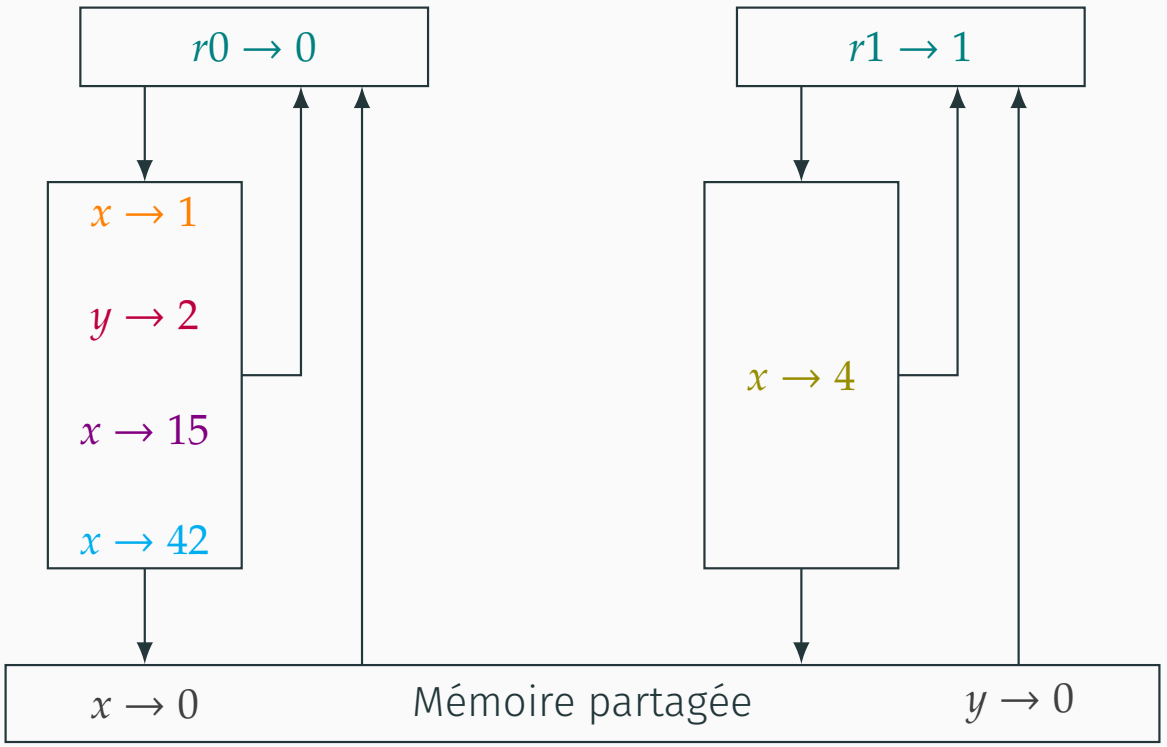
$x^{mem} = 0$

$y^{mem} = 1$

Encodage par variables — Partial Store Ordering

Une pseudo-variable par entrée du tampon.

L'ordre inter-variables est oublié par l'abstraction.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

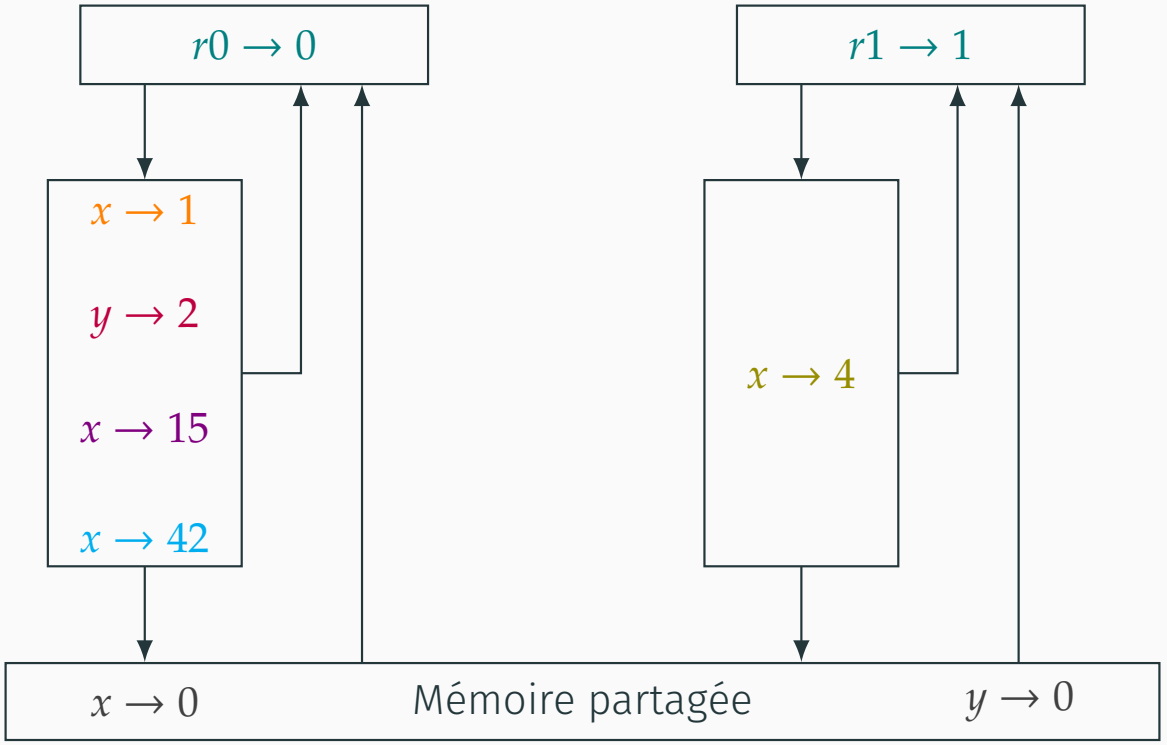
$x^{mem} = 0$

$y^{mem} = 1$

Encodage par variables — Partial Store Ordering

Une pseudo-variable par entrée du tampon.

L'ordre inter-variables est oublié par l'abstraction.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

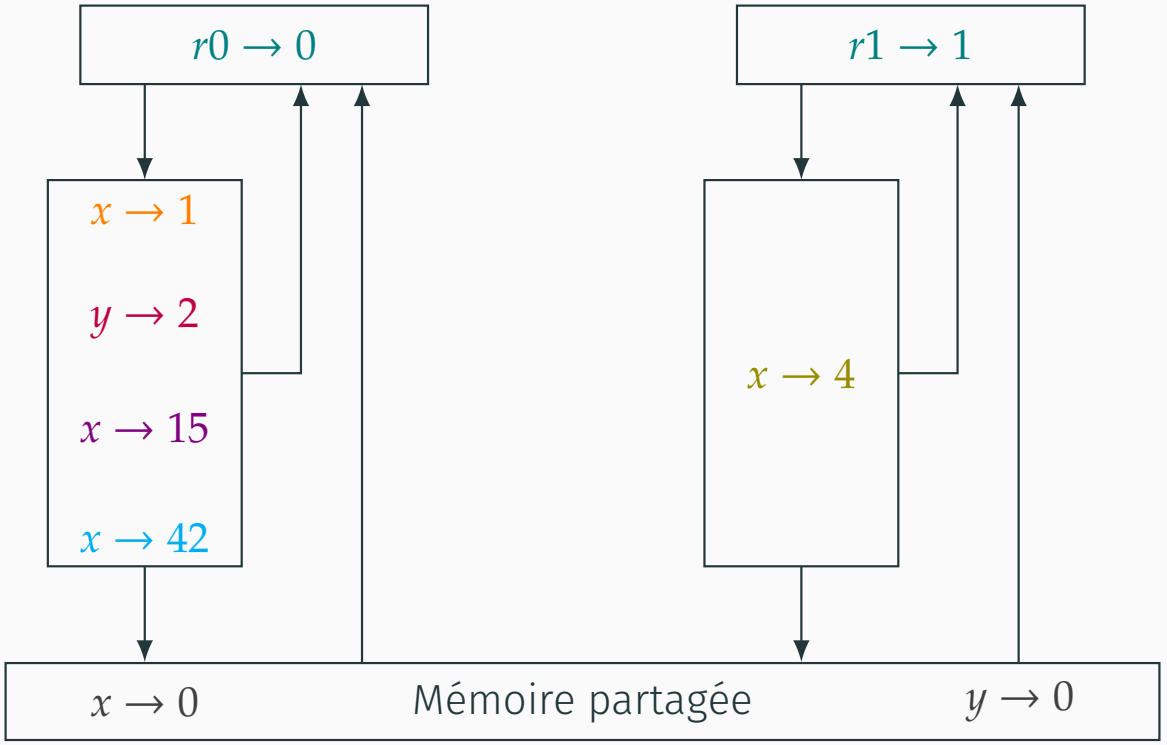
$x^{mem} = 0$

$y^{mem} = 1$

Encodage par variables — Partial Store Ordering

Une pseudo-variable par entrée du tampon.

L'ordre inter-variables est oublié par l'abstraction.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

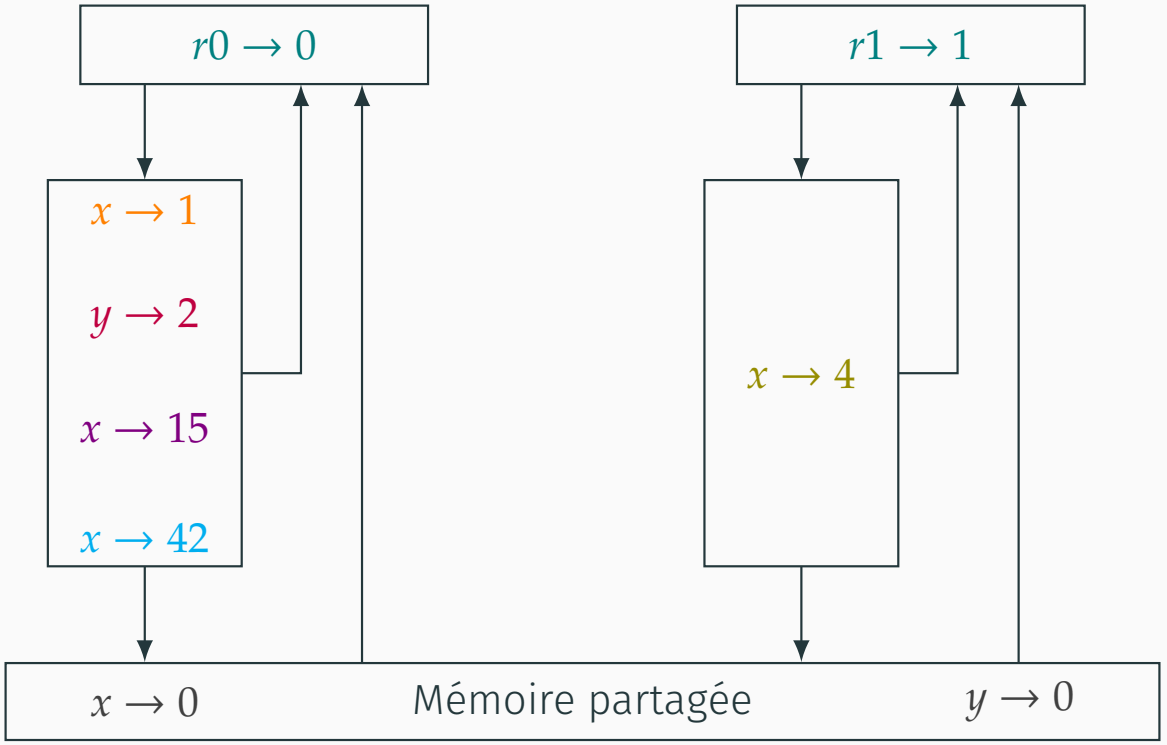
$x^{mem} = 0$

$y^{mem} = 1$

Encodage par variables — Partial Store Ordering

Une pseudo-variable par entrée du tampon.

L'ordre inter-variables est oublié par l'abstraction.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

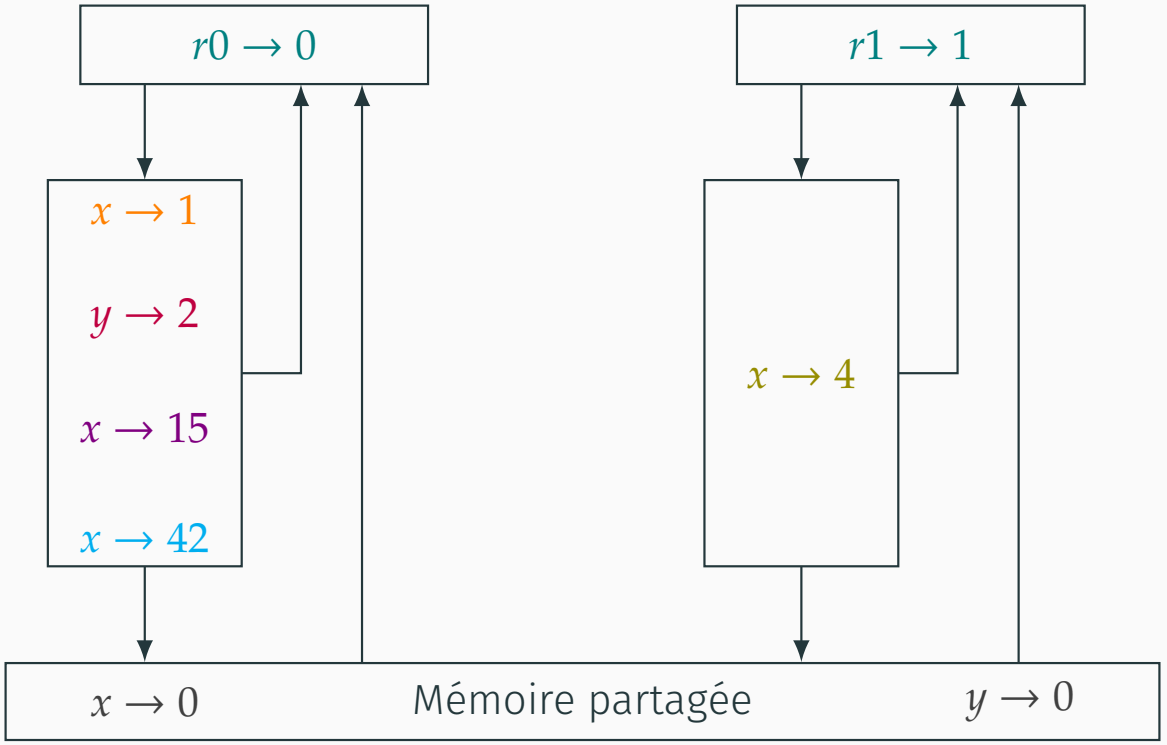
$x^{mem} = 0$

$y^{mem} = 1$

Encodage par variables — Partial Store Ordering

Une pseudo-variable par entrée du tampon.

L'ordre inter-variables est oublié par l'abstraction.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

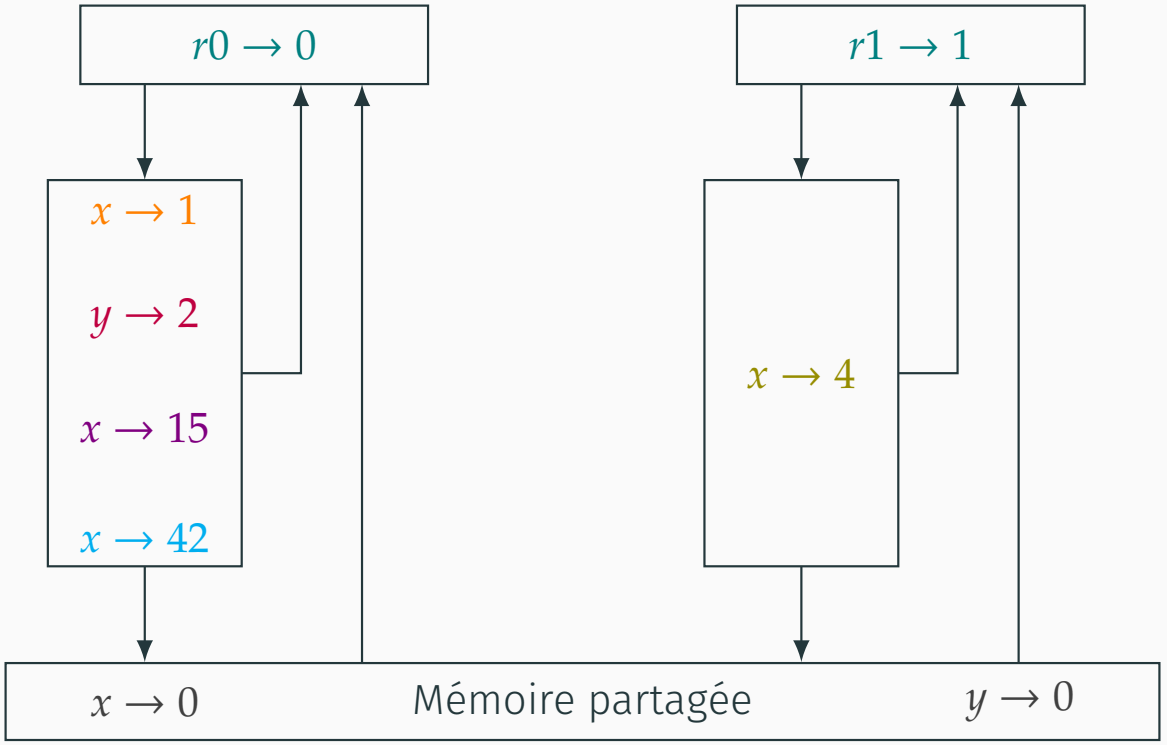
$x^{mem} = 0$

$y^{mem} = 1$

Encodage par variables — Partial Store Ordering

Une pseudo-variable par entrée du tampon.

L'ordre inter-variables est oublié par l'abstraction.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

$x^{mem} = 0$

$y^{mem} = 1$

Analyse monolithique

Partitionnement

Critère de partitionnement

Observation

Le comportement des opérations du programme diffère beaucoup si les tampons sont vides ou non.

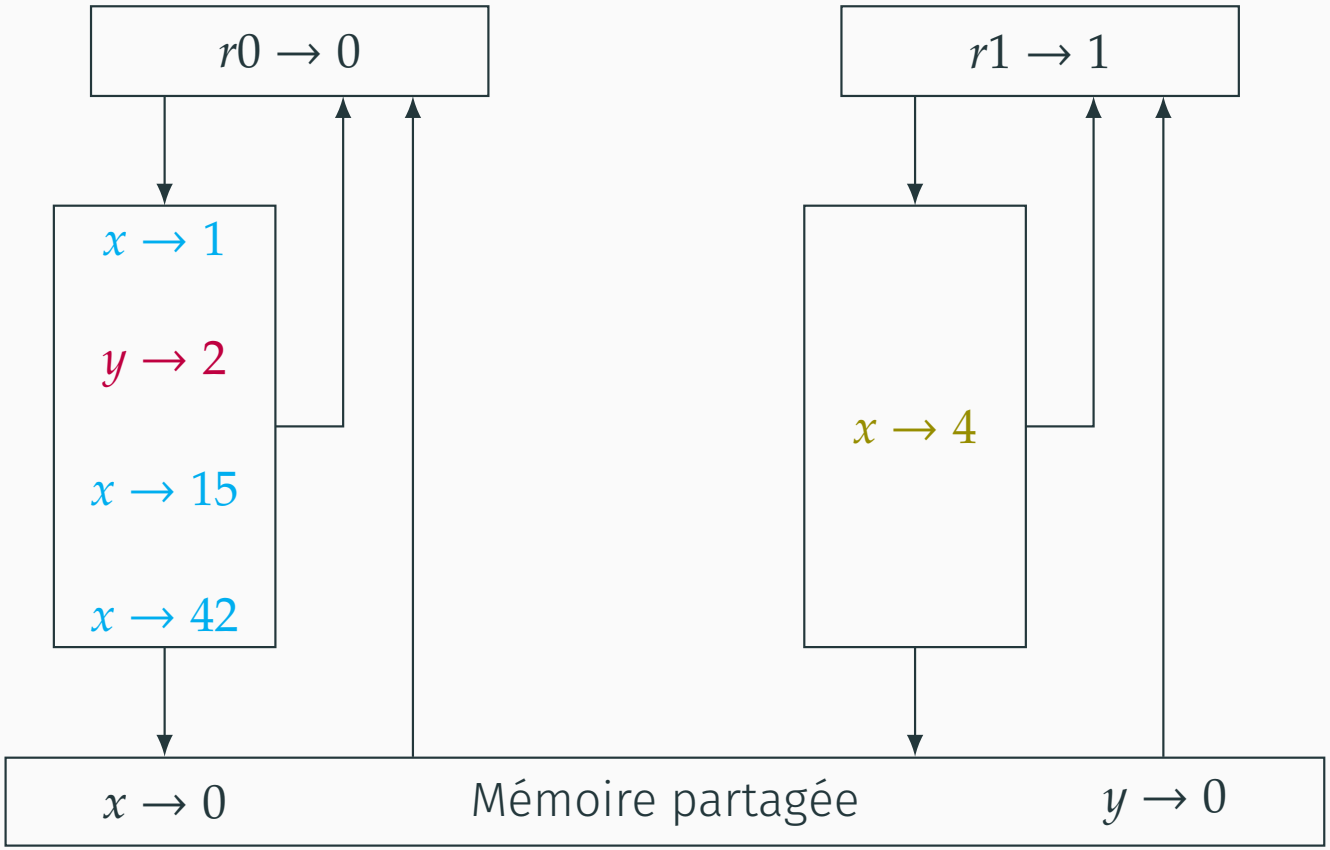
- On abstrait la taille des tampons : 0, 1 ou ω (plus de 1)
- Les états ayant des tampons de tailles abstraites identiques sont regroupés ensemble

Critère de partitionnement

Observation
Le comportement des opérations du programme diffère beaucoup si les tampons sont vides ou non.

- On abstrait la taille des tampons : 0, 1 ou ω (plus de 1)
- Les états ayant des tampons de tailles abstraites identiques sont regroupés ensemble

Tailles abstraites : exemple



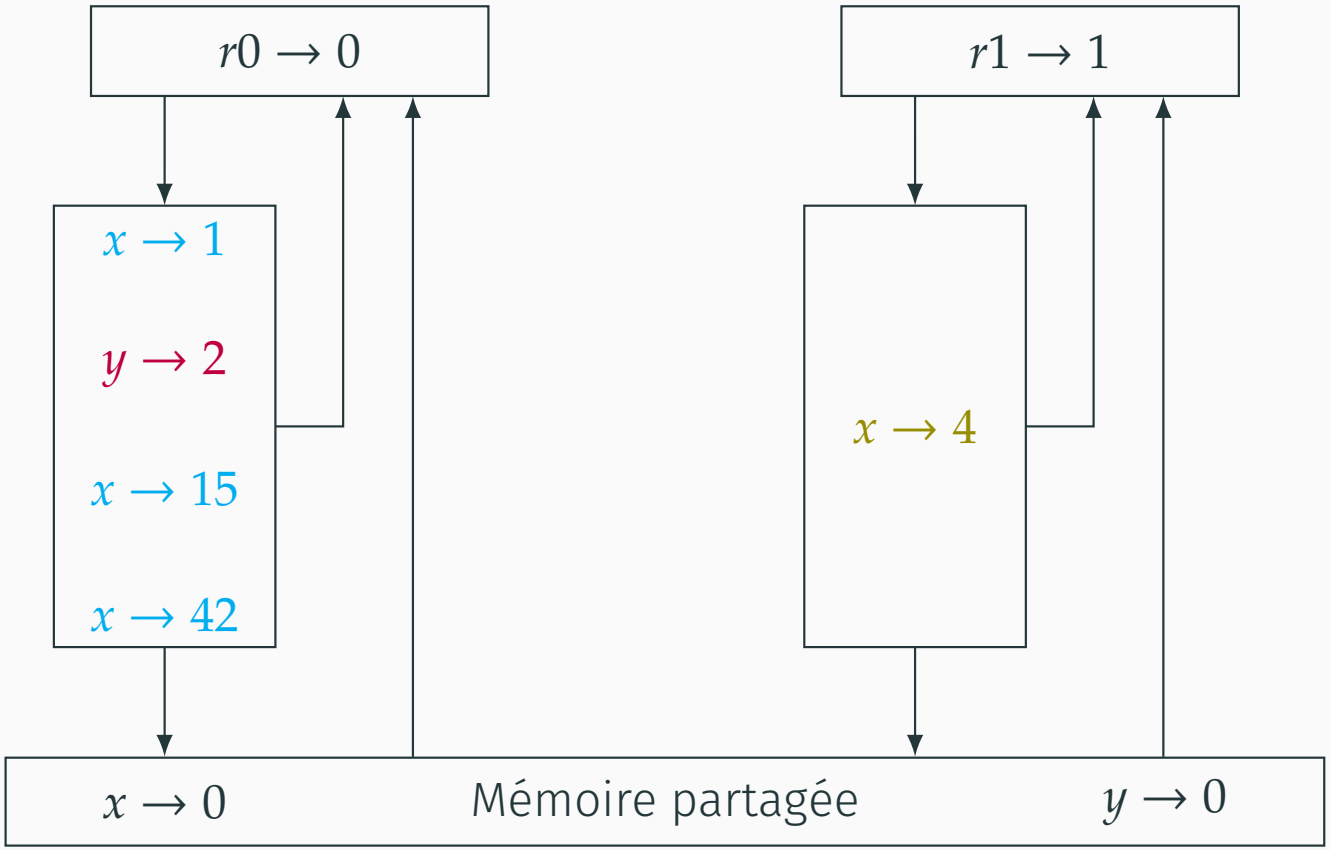
$(x, 0) \mapsto \omega$

$(y, 0) \mapsto 1$

$(x, 1) \mapsto 1$

$(y, 1) \mapsto 0$

Tailles abstraites : exemple



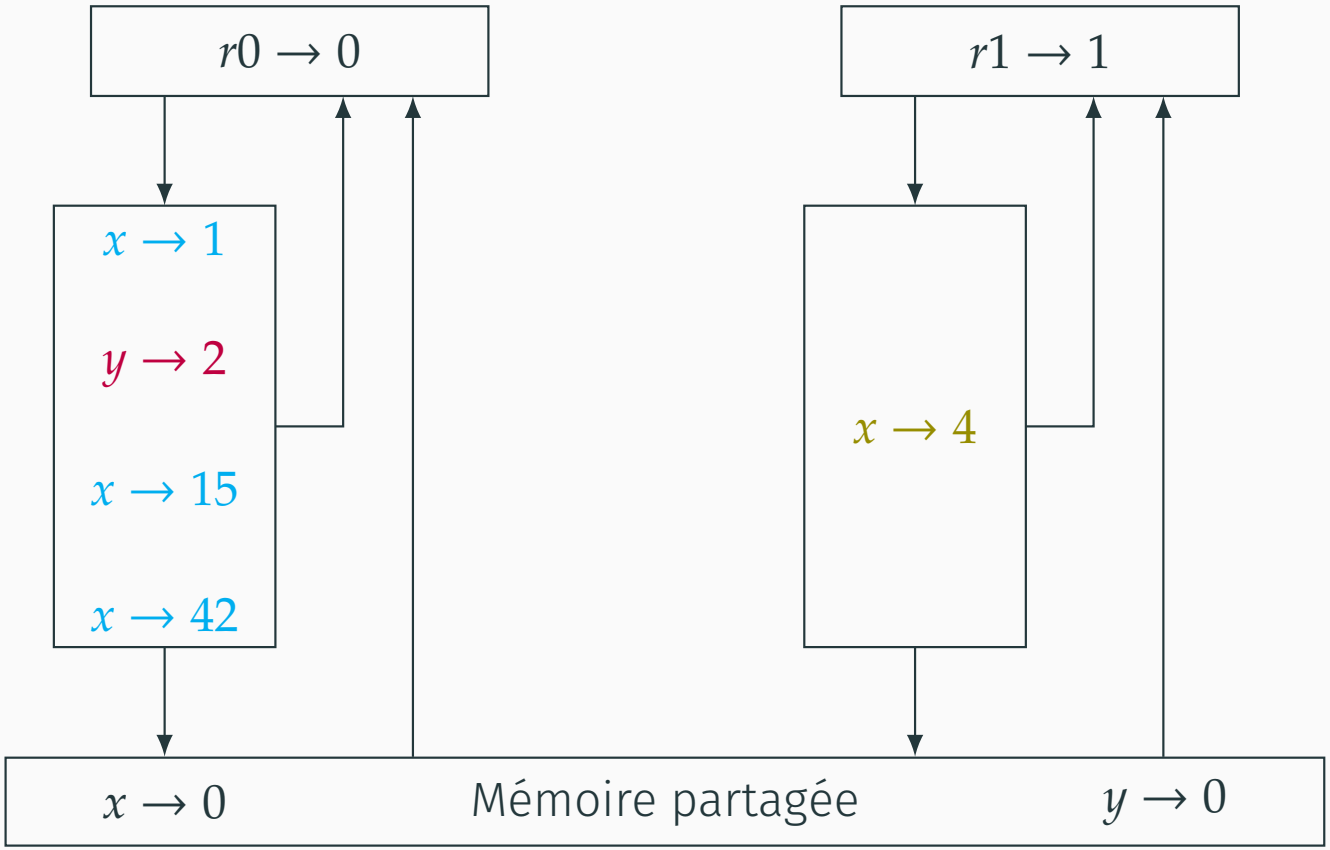
$(x, 0) \mapsto \omega$

$(y, 0) \mapsto 1$

$(x, 1) \mapsto 1$

$(y, 1) \mapsto 0$

Tailles abstraites : exemple



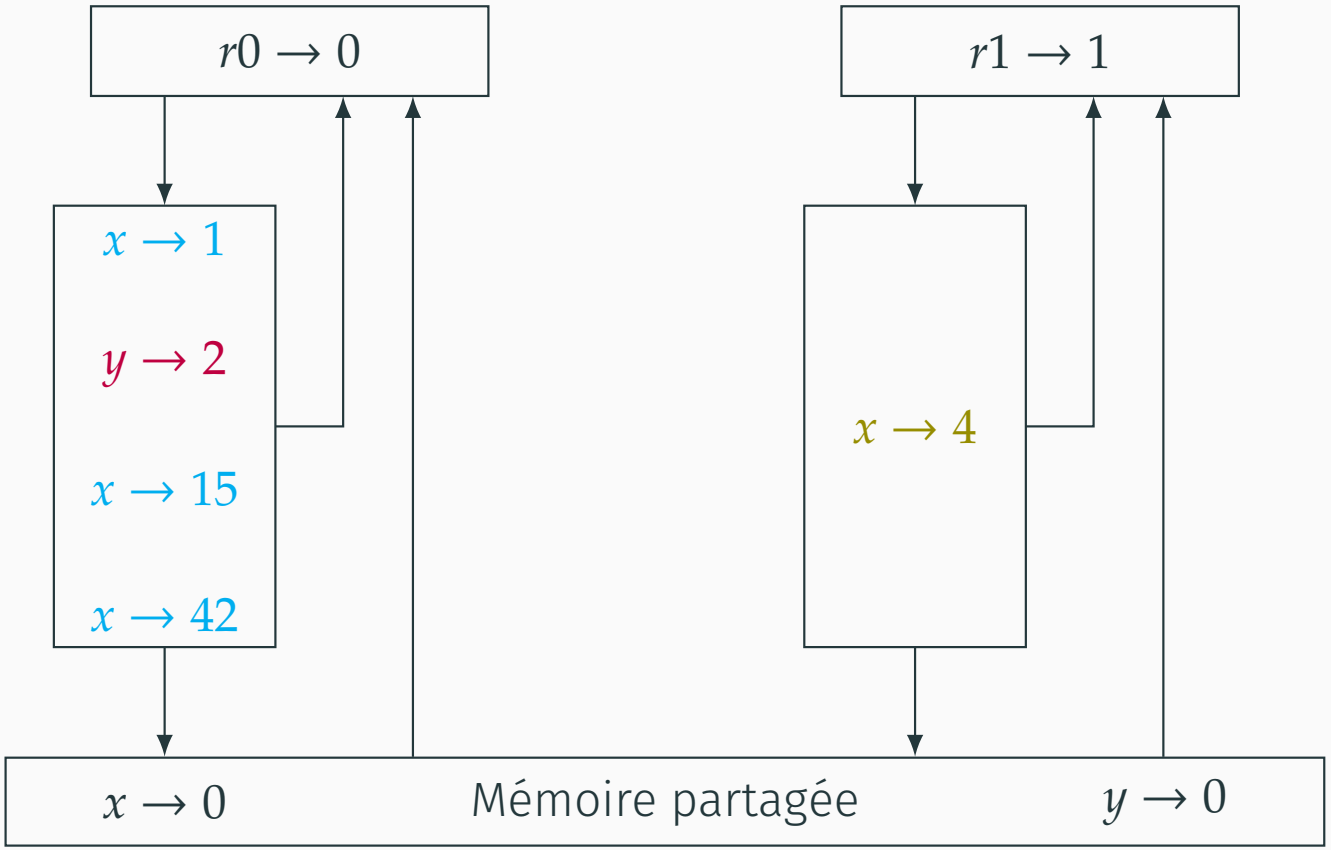
$(x, 0) \mapsto \omega$

$(y, 0) \mapsto 1$

$(x, 1) \mapsto 1$

$(y, 1) \mapsto 0$

Tailles abstraites : exemple



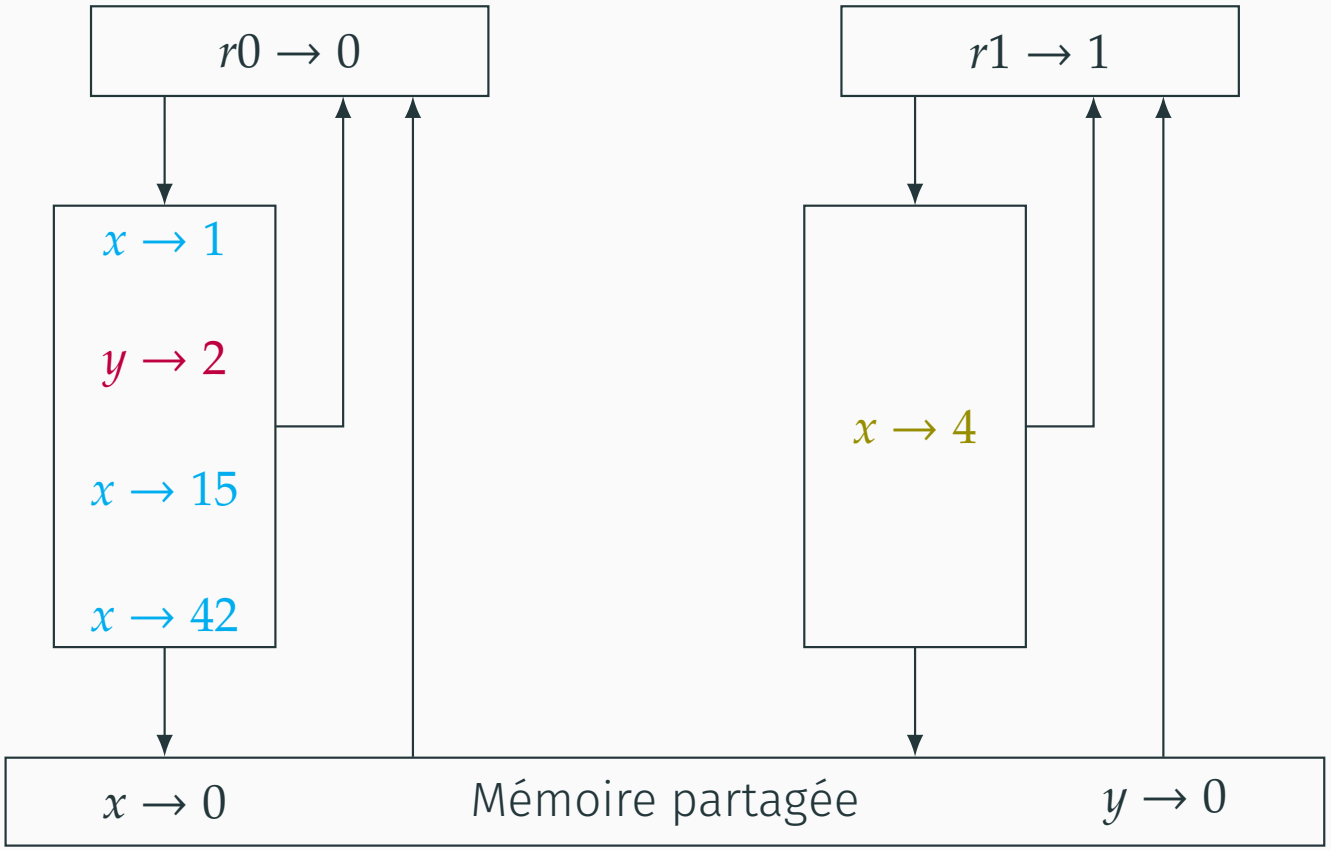
$(x, 0) \mapsto \omega$

$(y, 0) \mapsto 1$

$(x, 1) \mapsto 1$

$(y, 1) \mapsto 0$

Tailles abstraites : exemple



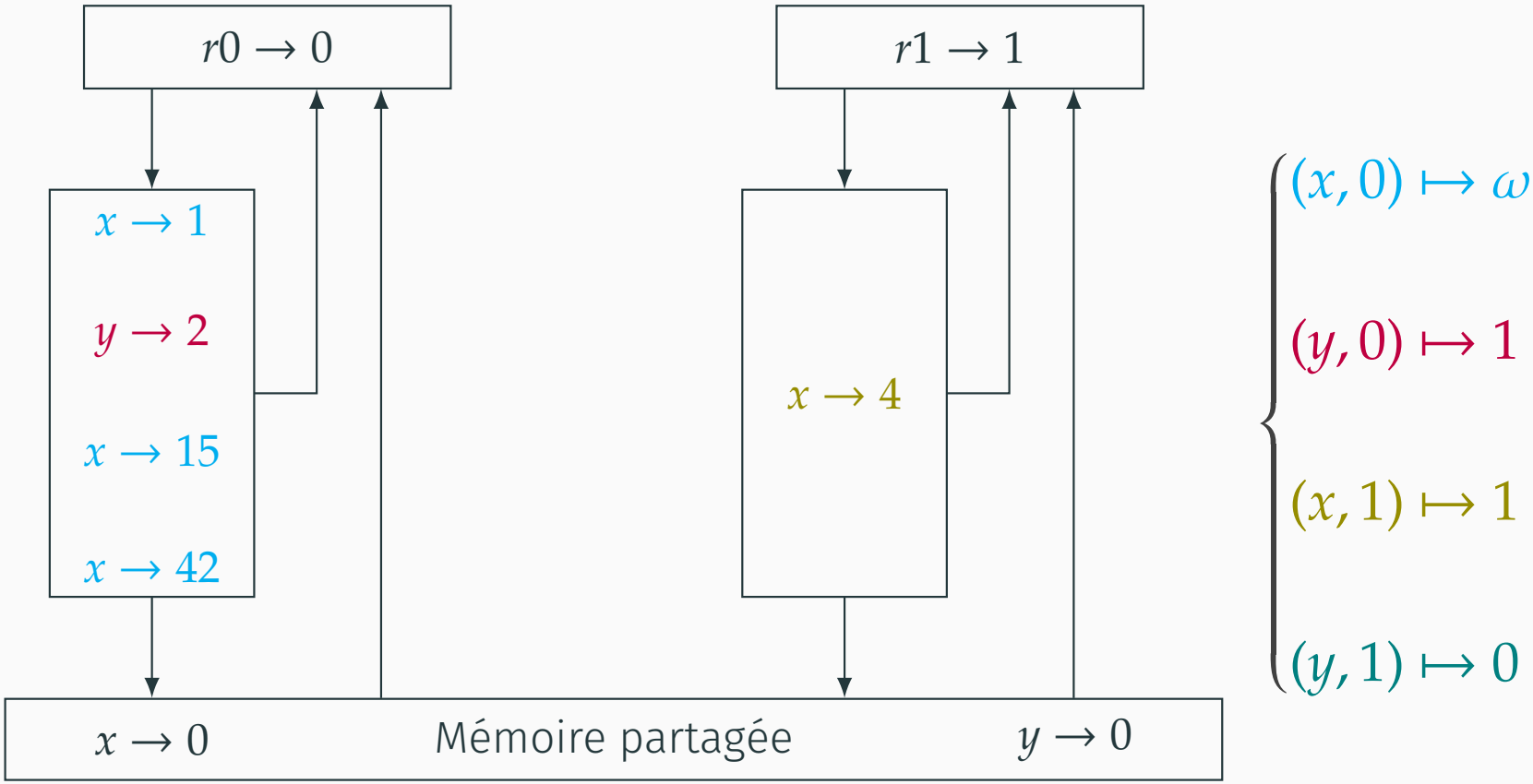
$(x, 0) \mapsto \omega$

$(y, 0) \mapsto 1$

$(x, 1) \mapsto 1$

$(y, 1) \mapsto 0$

Tailles abstraites : exemple



Analyse monolithique

Condensation

Condensation des tampons

Observation.

- Rôle spécifique de l'entrée la plus récente x_1^T : donne la valeur d'une lecture de x par T
- Les autres entrées ne serviront qu'à mettre à jour la mémoire

Condensation.

- On regroupe les variables $x_2^T, ..., x_\infty^T$ dans une seule variable condensée x_ω^T
- On distingue x_1^T pour maintenir la précision des lectures

Condensation des tampons

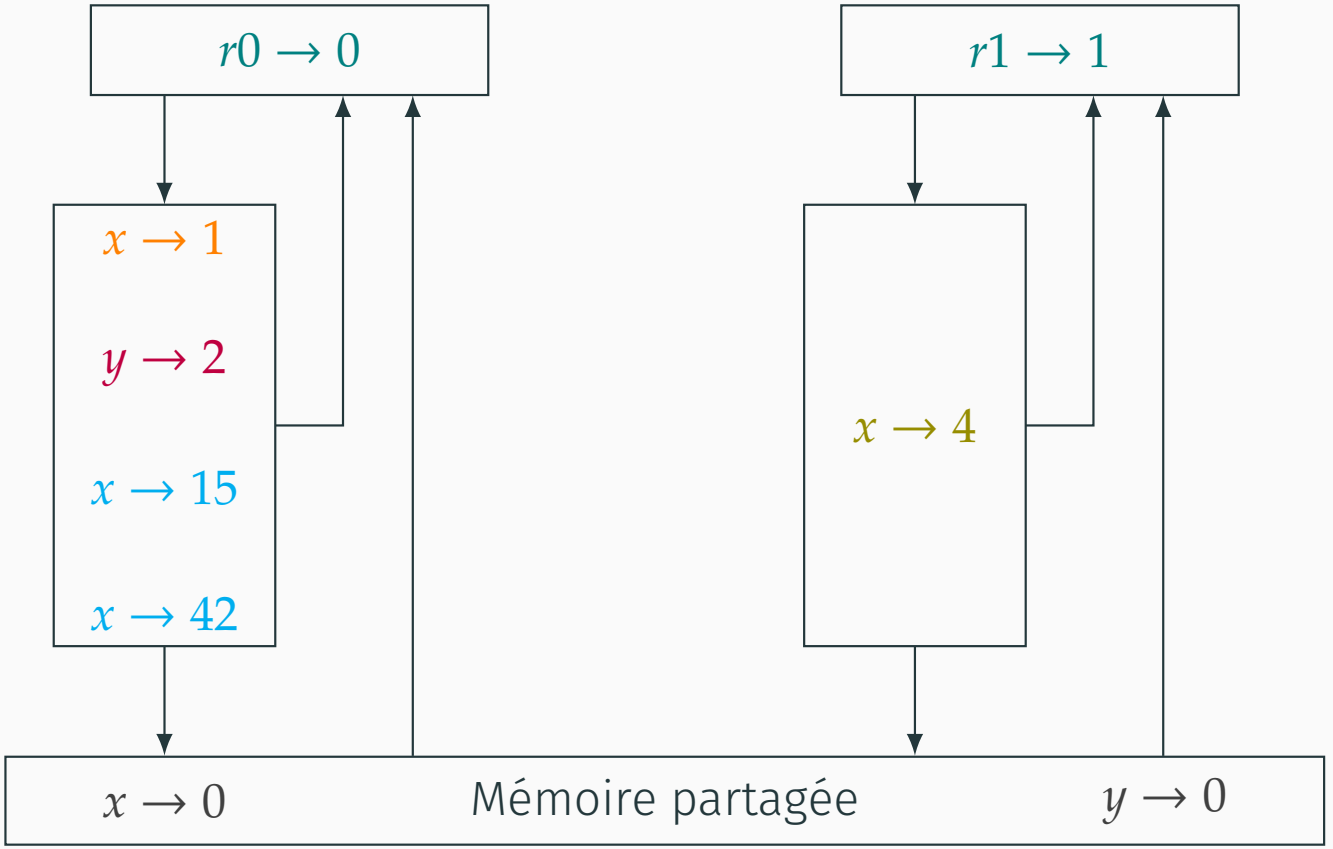
Observation.

- Rôle spécifique de l'entrée la plus récente x_1^T : donne la valeur d'une lecture de x par T
- Les autres entrées ne serviront qu'à mettre à jour la mémoire

Condensation.

- On regroupe les variables $x_2^T, ..., x_\infty^T$ dans une seule variable condensée x_ω^T
- On distingue x_1^T pour maintenir la précision des lectures

Condensation : exemple



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_\omega^0 \in \{15; 42\}$

$x_1^1 = 4$

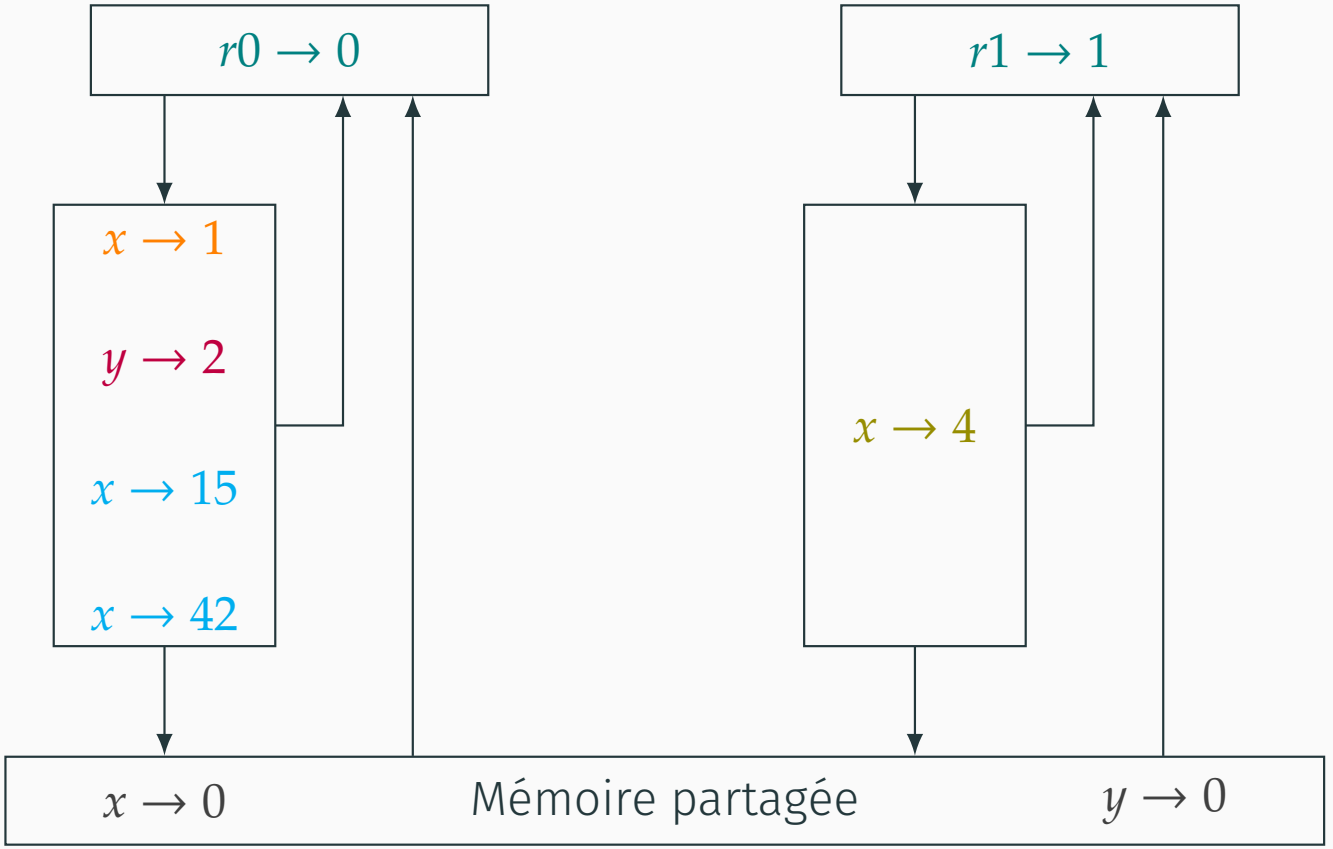
$x^{mem} = 0$

$y^{mem} = 1$

$(x, 0) \mapsto \omega$ $(y, 0) \mapsto 1$ $(x, 1) \mapsto 1$ $(y, 1) \mapsto 0$

x_1^0 x_ω^0 y_1^0 x_1^1

Condensation : exemple



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_\omega^0 \in \{15; 42\}$

$x_1^1 = 4$

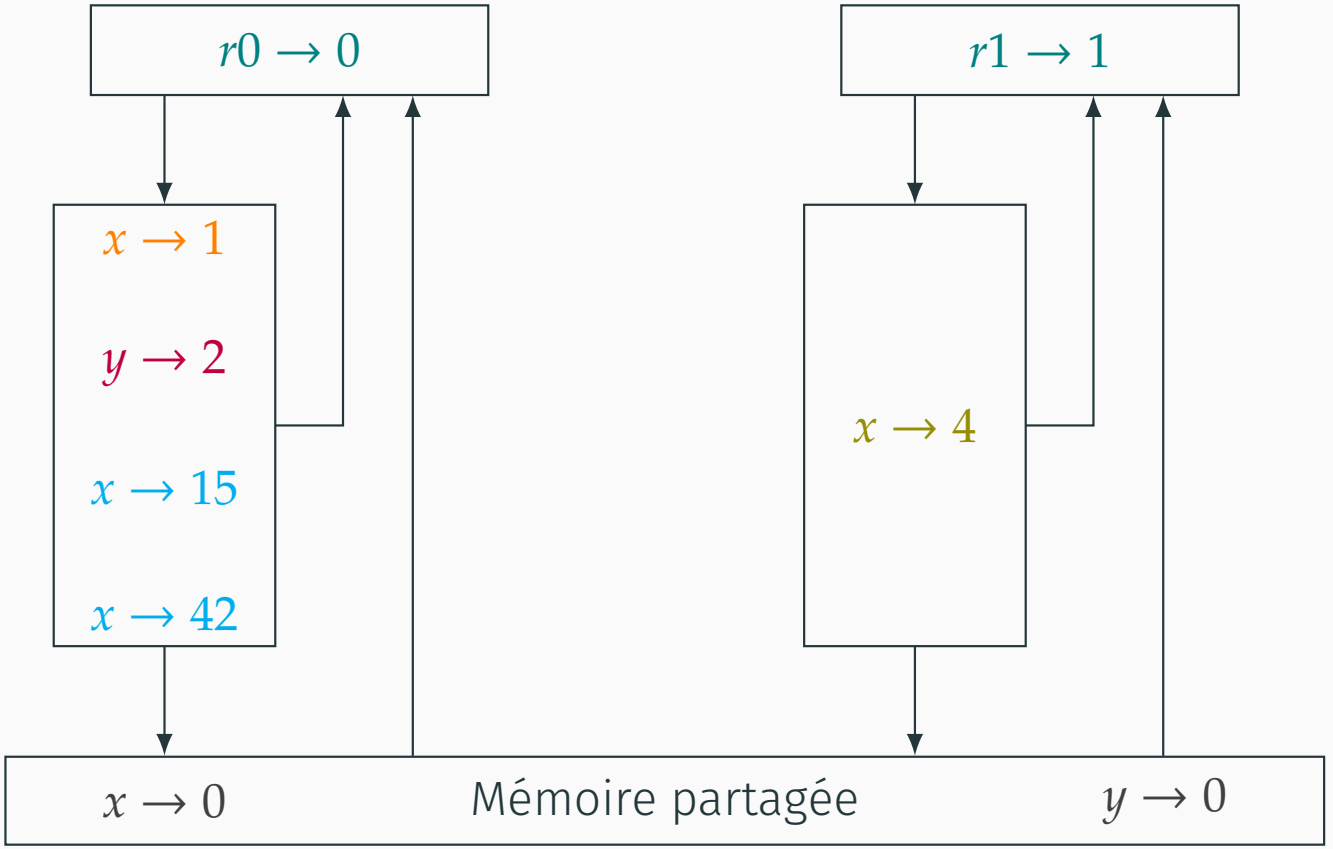
$x^{mem} = 0$

$y^{mem} = 1$

$(x, 0) \mapsto \omega$ $(y, 0) \mapsto 1$ $(x, 1) \mapsto 1$ $(y, 1) \mapsto 0$

x_1^0 x_ω^0 y_1^0 x_1^1

Condensation : exemple



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_\omega^0 \in \{15; 42\}$

$x_1^1 = 4$

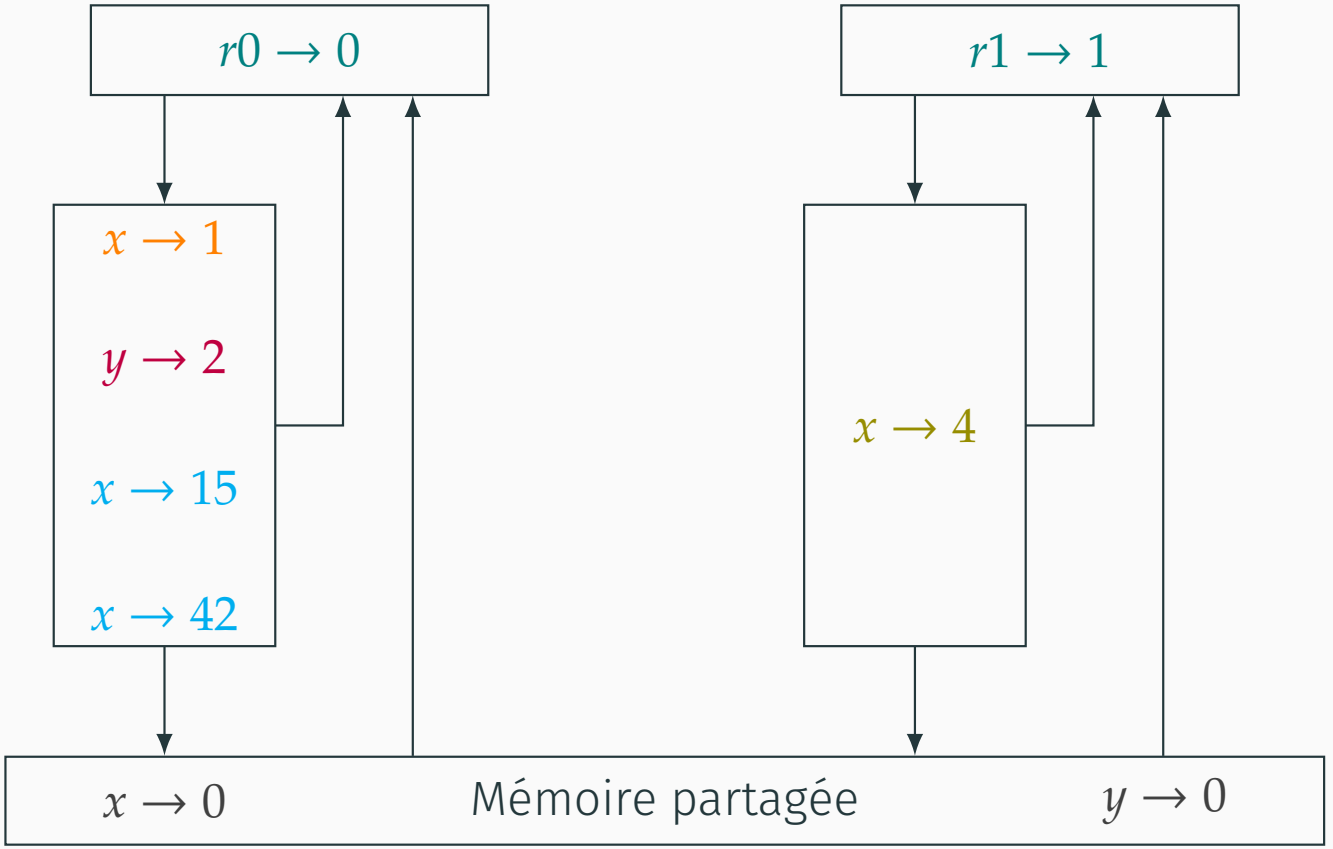
$x^{mem} = 0$

$y^{mem} = 1$

$(x, 0) \mapsto \omega$ $(y, 0) \mapsto 1$ $(x, 1) \mapsto 1$ $(y, 1) \mapsto 0$

x_1^0 x_ω^0 y_1^0 x_1^1

Condensation : exemple



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_\omega^0 \in \{15; 42\}$

$x_1^1 = 4$

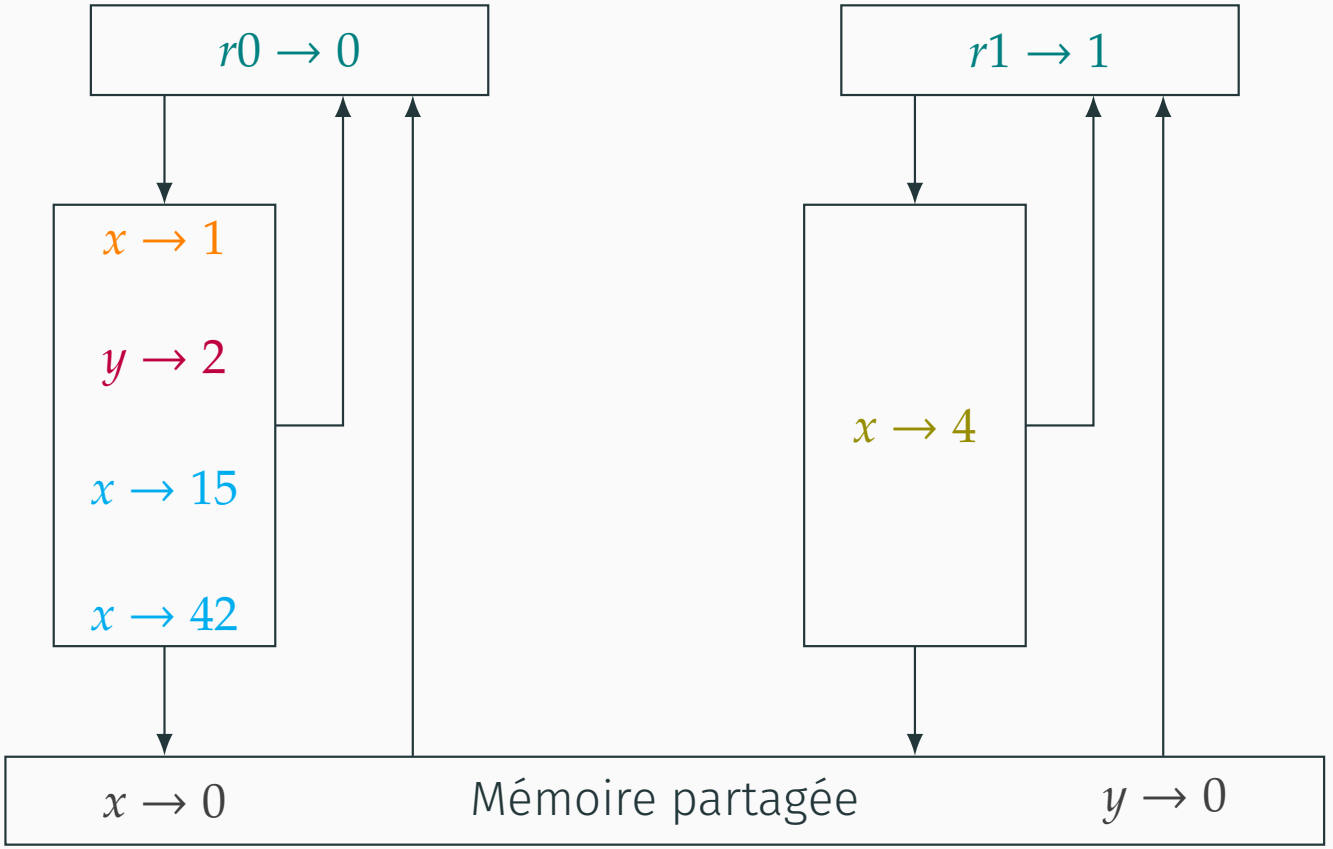
$x^{mem} = 0$

$y^{mem} = 1$

$(x, 0) \mapsto \omega$ $(y, 0) \mapsto 1$ $(x, 1) \mapsto 1$ $(y, 1) \mapsto 0$

x_1^0 x_ω^0 y_1^0 x_1^1

Condensation : exemple



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_\omega^0 \in \{15; 42\}$

$x_1^1 = 4$

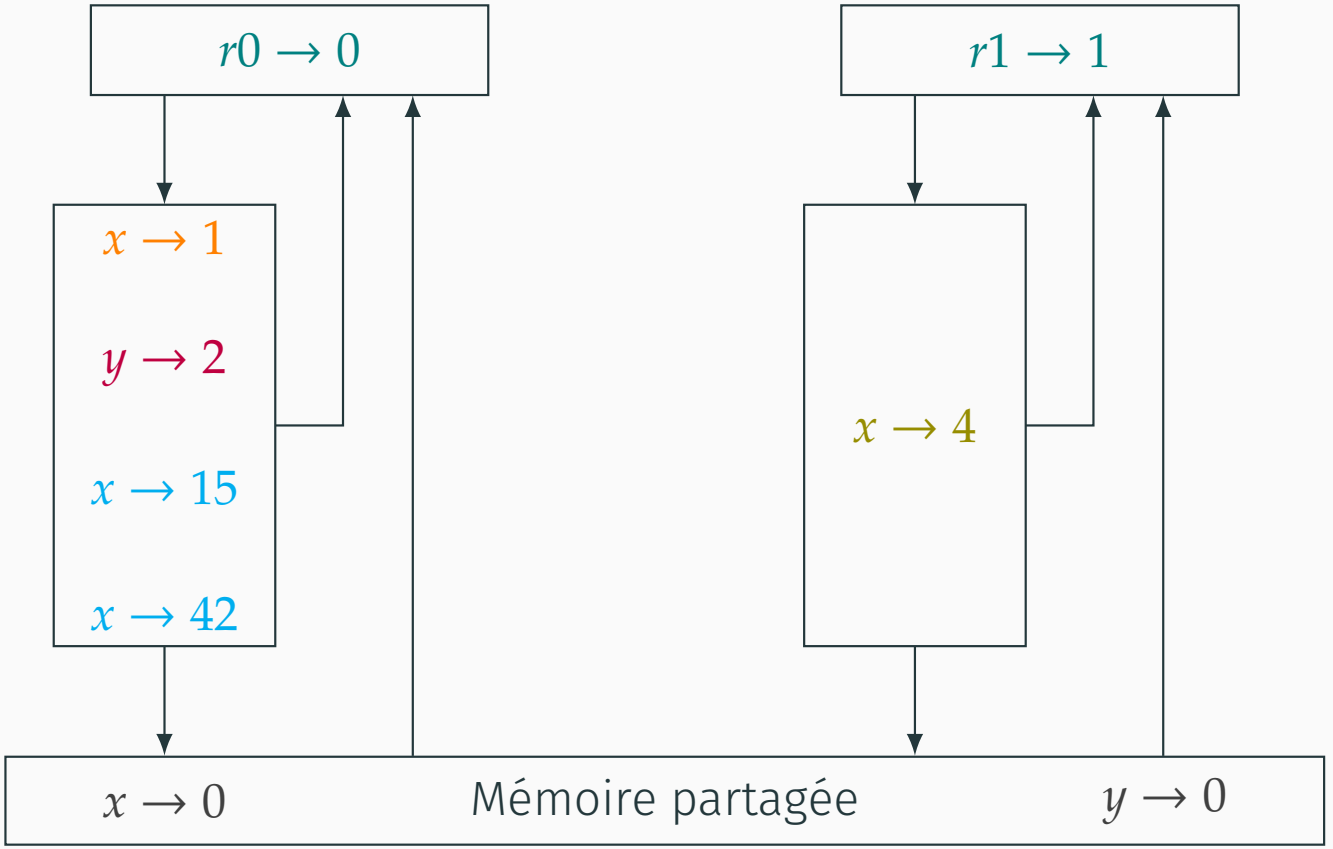
$x^{mem} = 0$

$y^{mem} = 1$

$(x, 0) \mapsto \omega$ $(y, 0) \mapsto 1$ $(x, 1) \mapsto 1$ $(y, 1) \mapsto 0$

x_1^0 x_ω^0 y_1^0 x_1^1

Condensation : exemple



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_\omega^0 \in \{15; 42\}$

$x_1^1 = 4$

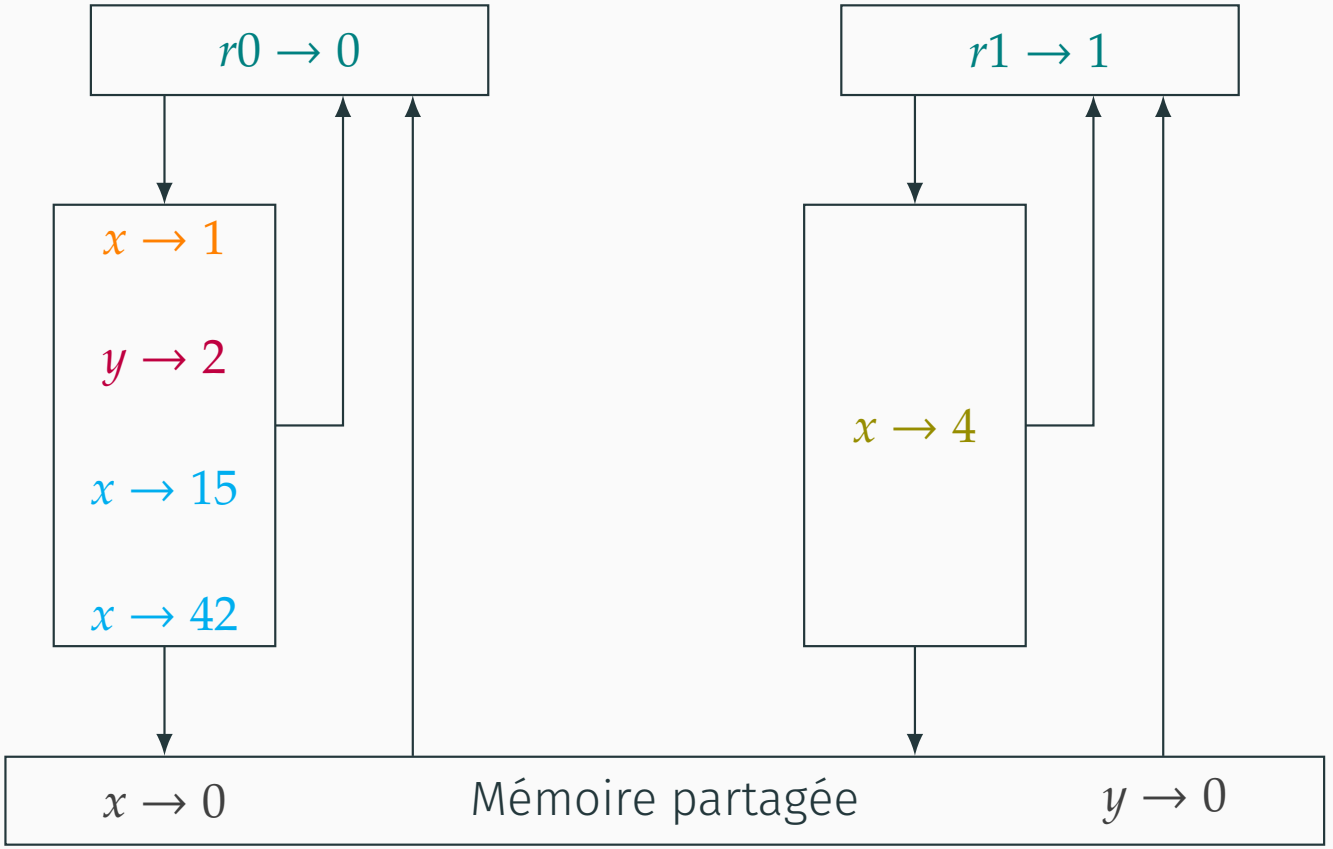
$x^{mem} = 0$

$y^{mem} = 1$

$(x, 0) \mapsto \omega$ $(y, 0) \mapsto 1$ $(x, 1) \mapsto 1$ $(y, 1) \mapsto 0$

x_1^0 x_ω^0 y_1^0 x_1^1

Condensation : exemple



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

$x_\omega^0 \in \{15; 42\}$

$x_1^1 = 4$

$x^{mem} = 0$

$y^{mem} = 1$

$(x,0) \mapsto \omega$ $(y,0) \mapsto 1$ $(x,1) \mapsto 1$ $(y,1) \mapsto 0$

x_1^0 x_ω^0 y_1^0 x_1^1

Analyse monolithique

Abstraction finale

Partitionnement et domaines numériques

- Après condensation, les états d'une même partition sont définis sur les mêmes variables
- On peut alors utiliser un domaine numérique pour abstraire chaque partition

Analyse monolithique

Résultats

Résultats expérimentaux

- Analyseur : 7k loc OCaml, Apron + BddApron
- Tests écrits dans un langage de modélisation spécifique

Algo *(nœuds)*	⟦mfence⟧	Tps (s)	⟦mfence⟧*	Tps (s)*
Abp *(100)*	0	0.3	0	6
Bakery *(400)*	-	-	4	3429
Concloop *(64)*	2	0.19	2	6
Dekker *(484)*	4	23	4	121
Kessel *(289)*	4	4	4	6
Loop2 TLM *(324)*	0	4.3	2	36
Peterson *(196)*	4	1.53	4	20
Queue *(54)*	0	0.15	1	1

* Dan, Meshman, Vechev, Yahav. Effective abstractions for verification under relaxed memory models. *VMCAI 2014.*

Analyse modulaire

Passage à l'échelle

L'analyse monolithique fonctionne bien pour 2 processus. Quid de 3 ? 5 ? 10 ? 100 ?

$\Longrightarrow$ Explosion combinatoire du graphe produit.

Les analyses monolithiques ne passent pas à l'échelle : il faut être modulaire.

Analyse modulaire

Interférences

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

(ℓ_0) $\bot$
(ℓ_1) $\bot$
(ℓ_2) $\bot$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0) $\bot$
(ℓ_1) $\bot$
(ℓ_2) $\bot$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

(ℓ_0) $x = 0$
(ℓ_1) $\bot$
(ℓ_2) $\bot$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0) $\bot$
(ℓ_1) $\bot$
(ℓ_2) $\bot$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

$(\ell_0) \quad x = 0$
$(\ell_1) \quad \bot$
$(\ell_2) \quad \bot$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

$(\ell_0) \quad x = 0$
$(\ell_1) \quad \bot$
$(\ell_2) \quad \bot$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

(ℓ_0) $x = 0$
(ℓ_1) $\bot$
(ℓ_2) $\bot$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0) $x = 0$
(ℓ_1) $x = 0$
(ℓ_2) $\bot$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

$$(\ell_0) \quad x = 0$$
$$(\ell_1) \quad \bot$$
$$(\ell_2) \quad \bot$$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

$$(\ell_0) \quad x = 0$$
$$(\ell_1) \quad x = 0$$
$$(\ell_2) \quad x = 1$$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

$$
\begin{array}{ll}
(\ell_0) & x = 0 \\
(\ell_1) & \bot \\
(\ell_2) & \bot
\end{array}
$$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

$$
\begin{array}{ll}
(\ell_0) & x = 0 \\
(\ell_1) & x = 0 \\
(\ell_2) & x = 1 \\
\text{Interférence} & x : 0 \mapsto 1
\end{array}
$$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

$$
\begin{array}{rl}
(\ell_0) & x = 0 \\
(\ell_1) & \bot \\
(\ell_2) & \bot
\end{array}
$$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

$$
\begin{array}{rl}
(\ell_0) & x \in \{0, 1\} \\
(\ell_1) & x = 0 \\
(\ell_2) & x = 1 \\
\text{Interférence} & x : 0 \mapsto 1
\end{array}
$$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

$$
\begin{array}{ll}
(\ell_0) & x = 0 \\
(\ell_1) & \bot \\
(\ell_2) & \bot
\end{array}
$$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

$$
\begin{array}{ll}
(\ell_0) & x \in \{0, 1\} \\
(\ell_1) & x = 0 \\
(\ell_2) & x = 1 \\
\text{Interférence} & x : 0 \mapsto 1
\end{array}
$$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

$$\begin{array}{ll} (\ell_0) & x = 0 \\ (\ell_1) & \bot \\ (\ell_2) & \bot \end{array}$$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

$$\begin{array}{ll} (\ell_0) & x \in \{0, 1\} \\ (\ell_1) & x = 0 \\ (\ell_2) & x = 1 \\ \text{Interférence} & x : 0 \mapsto 1 \end{array}$$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

(ℓ_0) $x \in \{0, 1\}$
(ℓ_1) $\bot$
(ℓ_2) $\bot$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0) $x \in \{0, 1\}$
(ℓ_1) $x = 0$
(ℓ_2) $x = 1$
Interférence $x : 0 \mapsto 1$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

(ℓ_0) $x \in \{0, 1\}$
(ℓ_1) $x = 1$
(ℓ_2) $\bot$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0) $x \in \{0, 1\}$
(ℓ_1) $x = 0$
(ℓ_2) $x = 1$
Interférence $x : 0 \mapsto 1$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 1$
(ℓ_2)	$x = 0$
Interférence	$x : 1 \mapsto 0$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Interférence	$x : 0 \mapsto 1$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 1$
(ℓ_2)	$x \in \{0, 1\}$
Interférence	$x : 1 \mapsto 0$

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x \in \{0, 1\}$
Interférence	$x : 0 \mapsto 1$

Une analyse modulaire en cohérence séquentielle

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Section critique ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Section critique ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x \in \{0,1\}$
(ℓ_1)	$x = 1$
(ℓ_2)	$x \in \{0,1\}$
Interférence	$x : 1 \mapsto 0$

(ℓ_0)	$x \in \{0,1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x \in \{0,1\}$
Interférence	$x : 0 \mapsto 1$

En $(\ell_1) \times (\ell_1) : x = 1 \wedge x = 0 \implies$ exclusion mutuelle.

Interférences

« Si T1 est en (ℓ_0), T2 en (ℓ_2), et si x vaut 3 et y vaut 4, alors $T1$ peut affecter 2 à x et ainsi passer en (ℓ_3) »

- Transitions : état d'origine $\mapsto$ état de destination
- Deux opérations :
  - Génération : par les opérations du programme
  - Application : arrive accessible $\Longrightarrow$ destination accessible
  - Générée par $T \Longrightarrow$ appliquée par $T' \neq T$
- Peuvent porter une information de contrôle
- On ajoute des variables auxiliaires pc_T aux états locaux

Interférences

« Si T1 est en (ℓ_0)*, T2 en* (ℓ_2)*, et si* x *vaut* 3 *et* y *vaut* 4*, alors* $T1$ *peut affecter* 2 *à* x *et ainsi passer en* (ℓ_3) *»*

- Transitions : état d'origine $\mapsto$ état de destination
- Deux opérations :
  - Génération : par les opérations du programme
  - Application : origine accessible $\implies$ destination accessible
  - Générée par T $\implies$ appliquée par $T' \neq T$
- Peuvent porter une information de contrôle
- On ajoute des variables auxiliaires pc_T aux états locaux

Interférences

« Si T1 est en (ℓ_0)*, T2 en* (ℓ_2)*, et si* x *vaut* 3 *et* y *vaut* 4*, alors* $T1$ *peut affecter* 2 *à* x *et ainsi passer en* (ℓ_3) *»*

- Transitions : état d'origine $\mapsto$ état de destination
- Deux opérations :
  - Génération : par les opérations du programme
  - Application : origine accessible $\implies$ destination accessible
  - Générée par T $\implies$ appliquée par $T' \neq T$
- Peuvent porter une information de contrôle
- On ajoute des variables auxiliaires pc_T aux états locaux

Interférences

« Si T1 est en (ℓ_0)*, T2 en* (ℓ_2)*, et si* x *vaut* 3 *et* y *vaut* 4*, alors* $T1$ *peut affecter* 2 *à* x *et ainsi passer en* (ℓ_3) *»*

- Transitions : état d'origine $\mapsto$ état de destination
- Deux opérations :
  - Génération : par les opérations du programme
  - Application : origine accessible $\implies$ destination accessible
  - Générée par T $\implies$ appliquée par $T' \neq T$
- Peuvent porter une information de contrôle
- On ajoute des variables auxiliaires pc_T aux états locaux

Interférences

« Si T1 est en (ℓ_0)*, T2 en* (ℓ_2)*, et si* x *vaut* 3 *et* y *vaut* 4*, alors* $T1$ *peut affecter* 2 *à* x *et ainsi passer en* (ℓ_3) *»*

- Transitions : état d'origine $\mapsto$ état de destination
- Deux opérations :
  - Génération : par les opérations du programme
  - Application : origine accessible $\implies$ destination accessible
  - Générée par T $\implies$ appliquée par $T' \neq T$
- Peuvent porter une information de contrôle
- On ajoute des variables auxiliaires pc_T aux états locaux

Interférences

« Si T1 est en (ℓ_0)*, T2 en* (ℓ_2)*, et si* x *vaut* 3 *et* y *vaut* 4*, alors* $T1$ *peut affecter* 2 *à* x *et ainsi passer en* (ℓ_3) *»*

- Transitions : état d'origine $\mapsto$ état de destination
- Deux opérations :
  - Génération : par les opérations du programme
  - Application : origine accessible $\implies$ destination accessible
  - Générée par T $\implies$ appliquée par $T' \neq T$
- Peuvent porter une information de contrôle
- On ajoute des variables auxiliaires pc_T aux états locaux

Interférences

« Si T1 est en (ℓ_0)*, T2 en* (ℓ_2)*, et si* x *vaut* 3 *et* y *vaut* 4*, alors* $T1$ *peut affecter* 2 *à* x *et ainsi passer en* (ℓ_3) *»*

- Transitions : état d'origine $\mapsto$ état de destination
- Deux opérations :
  - Génération : par les opérations du programme
  - Application : origine accessible $\implies$ destination accessible
  - Générée par T $\implies$ appliquée par $T' \neq T$
- Peuvent porter une information de contrôle
- On ajoute des variables auxiliaires pc_T aux états locaux

Interférences

« Si T1 est en (ℓ_0)*, T2 en* (ℓ_2)*, et si* x *vaut* 3 *et* y *vaut* 4*, alors* $T1$ *peut affecter* 2 *à* x *et ainsi passer en* (ℓ_3) *»*

- Transitions : état d'origine $\mapsto$ état de destination
- Deux opérations :
  - Génération : par les opérations du programme
  - Application : origine accessible $\implies$ destination accessible
  - Générée par T $\implies$ appliquée par $T' \neq T$
- Peuvent porter une information de contrôle
- On ajoute des variables auxiliaires pc_T aux états locaux

Analyse modulaire par interférences

- Point fixe imbriqué
  - Interne : analyse d'un processus paramétrée par un ensemble d'interférences des autres processus
  - Externe : stabilisation de la génération des interférences en répétant l'analyse interne
- Sémantique des interférences concrètes : sûre et complète

Analyse modulaire

Domaines abstraits

États locaux d'un processus T

- Une variable auxiliaire $pc_{T'}$ pour tout $T' \neq T$
- Oubli des tampons de tous les autres processus
- On garde toutes les variables locales
- On garde aussi la mémoire partagée
- Abstraction mémoire semblable au domaine monolithique

Abstraction du contrôle

Domaine des variables $pc_{T'}$.

Adapté de Raphaël Monat et Antoine Miné: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. *VMCAI 2017*.

3 choix possibles :

- Insensibilité au flot de contrôle : $\alpha(\ell) = \top$.
- Contrôle concret : $\alpha(\ell) = \ell$.
- Partitionnement du contrôle : on regroupe ensemble des points de contrôle choisis.

Abstraction du contrôle

Domaine des variables $pc_{T'}$.

Adapté de Raphaël Monat et Antoine Miné: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. *VMCAI 2017.*

3 choix possibles :

- Insensibilité au flot de contrôle : $\alpha(\ell) = \top$.
- Contrôle concret : $\alpha(\ell) = \ell$.
- Partitionnement du contrôle : on regroupe ensemble des points de contrôle choisis.

Abstraction du contrôle

Domaine des variables $pc_{T'}$.

Adapté de Raphaël Monat et Antoine Miné: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. *VMCAI 2017.*

3 choix possibles :

- Insensibilité au flot de contrôle : $\alpha(\ell) = \top$.
- Contrôle concret : $\alpha(\ell) = \ell$.
- Partitionnement du contrôle : on regroupe ensemble des points de contrôle choisis.

Abstraction du contrôle

Domaine des variables $pc_{T'}$.

Adapté de Raphaël Monat et Antoine Miné: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. *VMCAI 2017.*

3 choix possibles :

- Insensibilité au flot de contrôle : $\alpha(\ell) = \top$.
- Contrôle concret : $\alpha(\ell) = \ell$.
- Partitionnement du contrôle : on regroupe ensemble des points de contrôle choisis.

Abstraction du contrôle

Domaine des variables $pc_{T'}$.

Adapté de Raphaël Monat et Antoine Miné: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. *VMCAI 2017.*

3 choix possibles :

- Insensibilité au flot de contrôle : $\alpha(\ell) = \top$.
- Contrôle concret : $\alpha(\ell) = \ell$.
- Partitionnement du contrôle : on regroupe ensemble des points de contrôle choisis.

Partitionnement du contrôle : séparation aux points à vérifier

```
thread /* T1 */ {
    (ℓ0) while (ℓ1) true {
        (ℓ2) while (ℓ3) x != 1 {(ℓ4)} (ℓ5)
        /* Section critique ... */
        (ℓ6) x = 0; (ℓ7)
    } (ℓ8)
} (ℓ9)
```

$$\alpha((\ell_0)..(\ell_4)) = \ell_1^\sharp$$

$$\alpha((\ell_5)..(\ell_9)) = \ell_2^\sharp$$

Partitionnement du contrôle : séparation en tête de boucle

```
thread /* T1 */ {
    (ℓ0) while (ℓ1) true {
        (ℓ2) while (ℓ3) x != 1 {(ℓ4)} (ℓ5)
        /* Section critique ... */
        (ℓ6) x = 0; (ℓ7)
    } (ℓ8)
} (ℓ9)
```

$$\alpha((\ell_0)) = \ell_0^\sharp$$
$$\alpha((\ell_1)..(\ell_2)) = \ell_1^\sharp$$
$$\alpha((\ell_3)..(\ell_4)) = \ell_2^\sharp$$
$$\alpha((\ell_5)..(\ell_9)) = \ell_3^\sharp$$

Interférences abstraites

Construites avec l'information minimale commune à tous les états locaux :

- Oubli de tous les tampons
- Contrôle représenté dans les variables pc_T

Représentation de transitions par des variables *primées* :

$$x = 0 \qquad y = 1 \qquad pc_1 = (\ell_0) \qquad pc_2 = (\ell_2)$$
$$x' = 1 \qquad y' = 1 \qquad pc_1' = (\ell_2) \qquad pc_2' = (\ell_2)$$

Puis abstractions contrôle et mémoire similaires aux états locaux

Clôture par transfert : optimisation

Rappel : le transfert est non-déterministe $\implies$ résultats *clos*.

Clôture naïve

- À chaque étape, point fixe de $[\![flush]\!]$
- Les transferts peuvent découvrir de nouvelles interférences applicables, et il faut clore après leur application.

Clôture optimisée

- $[\![flush\ z]\!] \circ [\![x \leftarrow 1]\!] = [\![x \leftarrow 1]\!] \circ [\![flush\ z]\!]$
- Depuis un élément fermé, il suffit de calculer les transferts de x après les opérations qui lisent ou modifient x.

Clôture par transfert : optimisation

Rappel : le transfert est non-déterministe $\implies$ résultats *clos*.

Clôture naïve

- À chaque étape, point fixe de $[\![flush]\!]$
- Les transferts peuvent découvrir de nouvelles interférences applicables, et il faut clore après leur application.

Clôture optimisée

- $[\![flush\ z]\!] \circ [\![x \leftarrow 1]\!] = [\![x \leftarrow 1]\!] \circ [\![flush\ z]\!]$
- Depuis un élément fermé, il suffit de calculer les transferts de x après les opérations qui lisent ou modifient x.

Clôture par transfert : optimisation

Rappel : le transfert est non-déterministe $\implies$ résultats *clos*.

Clôture naïve

- À chaque étape, point fixe de $[\![flush]\!]$
- Les transferts peuvent découvrir de nouvelles interférences applicables, et il faut clore après leur application.

Clôture optimisée

- $[\![flush\ z]\!] \circ [\![x \leftarrow 1]\!] = [\![x \leftarrow 1]\!] \circ [\![flush\ z]\!]$
- Depuis un élément fermé, il suffit de calculer les transferts de x après les opérations qui lisent ou modifient x.

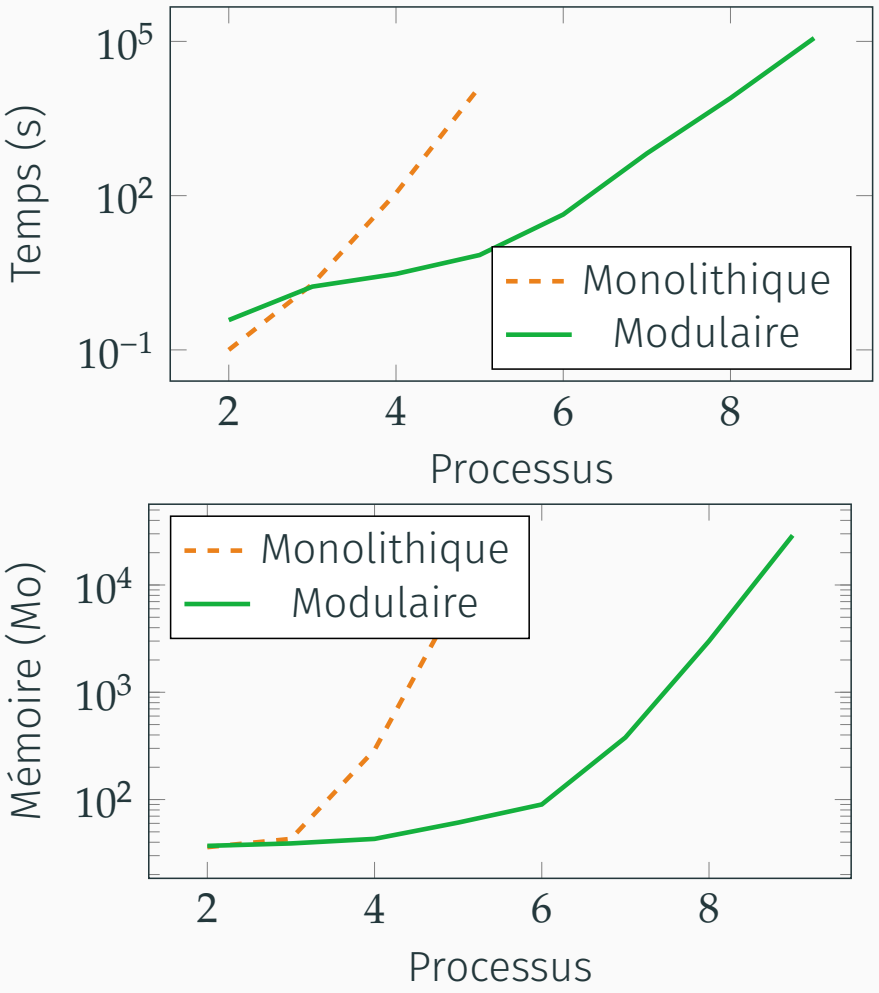
Analyse modulaire

Résultats

Précision

Test	Monolithique	Modulaire
abp	✓	✓
concloop	✓	✓
kessel	✓	✗
dekker	✓	✓
peterson	✓	✓
queue	✓	✓
bakery	⌛	✗

Passage à l'échelle




```
thread {
  while (x != 0) { };
  x = 1;
}

thread {
  while (x != 1) { };
  x = 2;
}
/* .......... */
thread {
  while (x != N) { };
  x = 0;
}
```

Abstraction avancées

Ordre inter-variables abstrait

- Notre abstraction des tampons oublie l'ordre entre deux variables différentes
- La condensation ne permet pas de l'ajouter directement :

$$x \to 1 \bullet x \to 2 \bullet y \to 3 \bullet x \to 4 \bullet y \to 5$$

$$y_0 \overset{?}{\gtrless} x_\omega$$

- Solution : ordre condensé partiel.

$$x_0 < y_0 \wedge x_\omega < y_\omega$$

Le transfert de x est alors impossible.

Ordre inter-variables et algorithme de Peterson

Barrière nécessaire dans PSO mais pas dans TSO.

```
/* Thread 0 */
flag_0 = true;
mfence;
turn = true;
mfence;

while (flag_1 && turn) { }
critical_section_thread0:
flag_0 = false;
```

```
/* Thread 1 */
flag_1 = true;
mfence;
turn = false;
mfence;

while (flag_0 && not turn) { }
critical_section_thread1:
flag_1 = false;
```

Abstraction non-uniforme : monotonie

Permet d'inférer *écritures croissantes* $\Longrightarrow$ *lectures croissantes*

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_2 = 4 \qquad x_3 = 3 \qquad x_4 = 2$$

$$\downarrow \alpha \downarrow$$

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_\omega = [2, 4]$$

$$\downarrow [\![flush\ x]\!]^\sharp \downarrow$$

$$x^{mem} = [2, 4] \qquad x_1 = 5 \qquad x_\omega = [2, 4]$$

Abstraction non-uniforme : monotonie

Permet d'inférer *écritures croissantes* $\Longrightarrow$ *lectures croissantes*

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_2 = 4 \qquad x_3 = 3 \qquad x_4 = 2$$

$$\downarrow \alpha \downarrow$$

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_\omega = [2,4] \qquad x_\omega \nearrow$$

$$\downarrow [\![flush\ x]\!]^\sharp \downarrow$$

$$x^{mem} = [2,4] \qquad x_1 = 5 \qquad x_\omega = [2,4]$$

Abstraction non-uniforme : monotonie

Permet d'inférer *écritures croissantes* $\Longrightarrow$ *lectures croissantes*

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_2 = 4 \qquad x_3 = 3 \qquad x_4 = 2$$

$$\downarrow \alpha \downarrow$$

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_\omega = [2,4] \qquad x_\omega \nearrow$$

$$\downarrow [\![flush\ x]\!]^\sharp \downarrow$$

$$x^{mem} = [2,4] \qquad x_1 = 5 \qquad x_\omega = [2,4] \qquad x_\omega \nearrow$$

Abstraction non-uniforme : monotonie

Permet d'inférer *écritures croissantes* $\Longrightarrow$ *lectures croissantes*

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_2 = 4 \qquad x_3 = 3 \qquad x_4 = 2$$

$$\downarrow \alpha \downarrow$$

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_\omega = [2,4] \qquad x_\omega \nearrow$$

$$\downarrow [\![flush\ x]\!]^\sharp \downarrow$$

$$x^{mem} = [2,4] \qquad x_1 = 5 \qquad x_\omega = [2,4] \qquad x_\omega \nearrow \qquad x^{mem} \leq x_\omega$$

Abstraction non-uniforme : monotonie

Permet d'inférer *écritures croissantes* $\Longrightarrow$ *lectures croissantes*

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_2 = 4 \qquad x_3 = 3 \qquad x_4 = 2$$

$$\downarrow \alpha \downarrow$$

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_\omega = [2,4] \qquad x_\omega \nearrow$$

$$\downarrow [\![flush\ x]\!]^\sharp \downarrow$$

$$x^{mem} = [2,4] \qquad x_1 = 5 \qquad x_\omega = [2,4] \qquad x_\omega \nearrow \qquad x^{mem} \leq x_\omega$$

Généralisation à l'aide d'une variable auxiliaire $x_{\omega+1}$:

$$x_{\omega+1} \leq x_\omega$$

Abstraction non-uniforme : monotonie

Permet d'inférer *écritures croissantes* $\Longrightarrow$ *lectures croissantes*

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_2 = 4 \qquad x_3 = 3 \qquad x_4 = 2$$

$$\downarrow \alpha \downarrow$$

$$x^{mem} = 0 \qquad x_1 = 5 \qquad x_\omega = [2, 4] \qquad x_\omega \nearrow$$

$$\downarrow [\![flush\ x]\!]^\sharp \downarrow$$

$$x^{mem} = [2, 4] \qquad x_1 = 5 \qquad x_\omega = [2, 4] \qquad x_\omega \nearrow \qquad x^{mem} \leq x_\omega$$

Généralisation à l'aide d'une variable auxiliaire $x_{\omega+1}$:

$$x_{\omega+1} \leq x_\omega$$

$$x_{\omega+1} = 2 \times x_\omega + 1$$

Conclusion

Conclusion

Réalisations

- Domaines abstraits pour l'analyse en cohérence faible
- Extension à l'analyse modulaire
- Implémentation et résultats encourageants
- Pistes d'amélioration de l'expressivité

Perspectives futures

- Autres modèles (Power/ARM, C, Java)
- Passage à l'échelle en production †
- On passe de 2 à 10, peut-on passer de 10 à N ? †

† Modularité requise !

Merci pour votre attention !