

# Dessine moi un domaine abstrait fini

## une recette à base de Camp4 et de solveurs SMT

---

Pierre Roux   Pierre-Loïc Garoche

17 janvier 2011

---

# Introduction

## **Avertissement**

Hormis pour l'enseignement de l'interprétation abstraite, ce que nous présentons aujourd'hui n'a guère d'applications.

# Introduction

## Avertissement

Hormis pour l'enseignement de l'interprétation abstraite, ce que nous présentons aujourd'hui n'a guère d'applications.

- La sémantique des programmes est généralement non calculable
- On veut donc en **calculer** une **sur-approximation**
- La théorie de l'interprétation abstraite fournit un cadre pour effectuer de tels calculs

# Introduction

## Avertissement

Hormis pour l'enseignement de l'interprétation abstraite, ce que nous présentons aujourd'hui n'a guère d'applications.

- La sémantique des programmes est généralement non calculable
- On veut donc en **calculer** une **sur-approximation**
- La théorie de l'interprétation abstraite fournit un cadre pour effectuer de tels calculs
- On cherche à générer une partie de tels analyseurs :
  - description de la sur-approximation (structure de treillis)
  - sémantique d'opérateurs de base (+, -, \*)

- 1 Interprétation abstraite
- 2 Construction du treillis
- 3 Opérateurs abstraits
- 4 Résultats

# Treillis

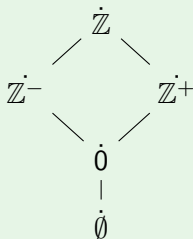
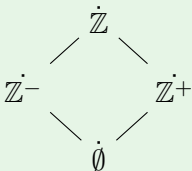
## Définition

**Treillis complet** : ensemble  $A$  muni de :

- ordre partiel  $\sqsubseteq$
- borne supérieure  $\bigsqcup$  : pour toute partie  $S$  de  $A$ ,  
 $\forall x \in S, x \sqsubseteq \bigsqcup S$  (majorant)  
 $\forall y \in A, (\forall x \in S, x \sqsubseteq y) \Rightarrow \bigsqcup S \sqsubseteq y$  (le plus petit)

## Exemple

Deux treillis complet



# Borne inférieure et infimums

## Propriété

Tout treillis complet  $A$  possède :

- borne inférieure  $\bigcap : S \mapsto \bigcap \{y \mid \forall x \in S, y \sqsubseteq x\}$
- plus petit élément :  $\perp = \bigcap \emptyset = \bigcap A$
- plus grand élément :  $\top = \bigcup A = \bigcap \emptyset$

# Borne inférieure et infimums

## Propriété

Tout treillis complet  $A$  possède :

- borne inférieure  $\prod : S \mapsto \bigsqcup \{y \mid \forall x \in S, y \sqsubseteq x\}$
- plus petit élément :  $\perp = \bigsqcup \emptyset = \prod A$
- plus grand élément :  $\top = \bigsqcup A = \prod \emptyset$

## Exemple : parties d'un ensemble

Pour tout ensemble (ex.  $\mathbb{Z}$ ), l'**ensemble de ses parties** ( $\mathcal{P}(\mathbb{Z})$ ) muni de l'**ordre inclusion**  $\sqsubseteq = \subseteq$  et de la **borne supérieure union**  $\sqcup = \cup$  est un treillis complet et :

- $\prod = \cap$
- $\perp = \emptyset$
- $\top = \mathbb{Z}$



# Fonction de concrétisation

## Définitions

- **concret** : treillis complet  $(S, \sqsubseteq, \sqcup)$   
(ex.  $(\mathcal{P}(\mathbb{Z}), \subseteq, \cup)$  pour une variable entière)
- **abstrait** : treillis complet  $(S^\#, \sqsubseteq^\#, \sqcup^\#)$
- **fonction de concrétisation** :  $\gamma : S^\# \rightarrow S$   
( $s^\# \in S^\#$  est une sur-approximation de  $s \in S$  si  $s \sqsubseteq \gamma(s^\#)$ )

## Exemple

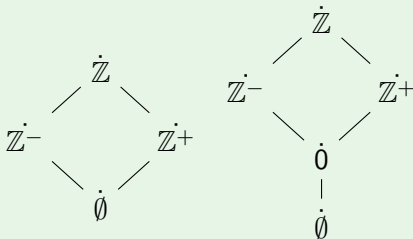
$$\gamma(\dot{\mathbb{Z}}) = \mathbb{Z}$$

$$\gamma(\dot{\mathbb{Z}}^-) = \{x \in \mathbb{Z} \mid x \leq 0\}$$

$$\gamma(\dot{\mathbb{Z}}^+) = \{x \in \mathbb{Z} \mid x \geq 0\}$$

$$\gamma(\dot{0}) = \{0\}$$

$$\gamma(\dot{\emptyset}) = \emptyset$$



# Opérateurs abstraits

## Définition

Pour une opération  $f$  du programme concret  
(ex. l'addition  $+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ ),  
 $f^\sharp$  est une **abstraction correcte** de  $f$  si :

$$\forall x_1^\sharp, \dots, x_n^\sharp \in S^\sharp, f \left( \gamma(x_1^\sharp), \dots, \gamma(x_n^\sharp) \right) \sqsubseteq \gamma \left( f^\sharp \left( x_1^\sharp, \dots, x_n^\sharp \right) \right)$$

# Opérateurs abstraits

## Définition

Pour une opération  $f$  du programme concret  
(ex. l'addition  $+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ ),  
 $f^\#$  est une **abstraction correcte** de  $f$  si :

$$\forall x_1^\#, \dots, x_n^\# \in S^\#, f \left( \gamma(x_1^\#), \dots, \gamma(x_n^\#) \right) \sqsubseteq \gamma \left( f^\# \left( x_1^\#, \dots, x_n^\# \right) \right)$$

## Exemples

- $\dot{0} +^\# \dot{0} = \dot{0}$
- $\dot{\mathbb{Z}}^+ +^\# \dot{\mathbb{Z}}^+ = \dot{\mathbb{Z}}^+$
- $\dot{\mathbb{Z}}^+ +^\# \dot{\mathbb{Z}}^- = \dot{\mathbb{Z}}$
- ...

# Opérateurs abstraits

## Définition

Pour une opération  $f$  du programme concret  
(ex. l'addition  $+$  :  $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ ),  
 $f^\#$  est une **abstraction correcte** de  $f$  si :

$$\forall x_1^\#, \dots, x_n^\# \in S^\#, f \left( \gamma(x_1^\#), \dots, \gamma(x_n^\#) \right) \sqsubseteq \gamma \left( f^\# \left( x_1^\#, \dots, x_n^\# \right) \right)$$

## Exemples

- $\dot{0} +^\# \dot{0} = \dot{0}$
- $\dot{\mathbb{Z}}^+ +^\# \dot{\mathbb{Z}}^+ = \dot{\mathbb{Z}}^+$        $+^\# : . \mapsto \dot{\mathbb{Z}}$
- $\dot{\mathbb{Z}}^+ +^\# \dot{\mathbb{Z}}^- = \dot{\mathbb{Z}}$       est toujours correct
- ...

# Meilleure abstraction

- On suppose  $\gamma$  monotone (si  $s^\sharp \sqsubseteq^\sharp s'^\sharp$  alors  $\gamma(s^\sharp) \sqsubseteq \gamma(s'^\sharp)$ )
- l'ordre  $\sqsubseteq^\sharp$  signifie donc « est plus précis que »

# Meilleure abstraction

- On suppose  $\gamma$  monotone (si  $s^\# \sqsubseteq^\# s'^\#$  alors  $\gamma(s^\#) \sqsubseteq \gamma(s'^\#)$ )
- l'ordre  $\sqsubseteq^\#$  signifie donc « est plus précis que »

## Définition

$s^\# \in S^\#$  est la **meilleure abstraction** de  $s \in S$  s'il est plus petit que toutes les autres sur-approximations de  $s$ .

# Meilleure abstraction

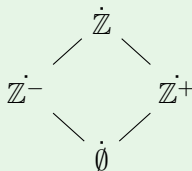
- On suppose  $\gamma$  monotone (si  $s^\# \sqsubseteq^\# s'^\#$  alors  $\gamma(s^\#) \sqsubseteq \gamma(s'^\#)$ )
- l'ordre  $\sqsubseteq^\#$  signifie donc « est plus précis que »

## Définition

$s^\# \in S^\#$  est la **meilleure abstraction** de  $s \in S$  s'il est plus petit que toutes les autres sur-approximations de  $s$ .

## Existence

On n'a pas toujours existence :  
 $\mathbb{Z}^-$  et  $\mathbb{Z}^+$  sont des sur-approximations de 0 incomparables



# Correspondance de Galois

## Définition

Soit  $(S, \sqsubseteq, \cup)$  et  $(S^\#, \sqsubseteq^\#, \cup^\#)$  deux treillis complet.  
Deux fonctions  $\alpha : S \rightarrow S^\#$  et  $\gamma : S^\# \rightarrow S$  forment une **correspondance de Galois** si :

$$\forall s \in S, \forall s^\# \in S^\#, \alpha(s) \sqsubseteq^\# s^\# \Leftrightarrow s \sqsubseteq \gamma(s^\#)$$



# Correspondance de Galois

## Définition

Soit  $(S, \sqsubseteq, \cup)$  et  $(S^\#, \sqsubseteq^\#, \cup^\#)$  deux treillis complet.  
Deux fonctions  $\alpha : S \rightarrow S^\#$  et  $\gamma : S^\# \rightarrow S$  forment une **correspondance de Galois** si :

$$\forall s \in S, \forall s^\# \in S^\#, \alpha(s) \sqsubseteq^\# s^\# \Leftrightarrow s \sqsubseteq \gamma(s^\#)$$

## Théorème

L'existence d'une meilleure abstraction pour tout élément de  $S$  équivaut à l'existence d'une fonction  $\alpha$  formant une correspondance de Galois avec  $\gamma$ .

La meilleure abstraction d'une opération  $f$  dans  $S$  est alors :

$$f^\# = \alpha \circ f \circ \gamma$$

1 Interprétation abstraite

**2 Construction du treillis**

3 Opérateurs abstraits

4 Résultats

# Signature du module OCaml

```
module type Lattice = sig  
  type t  
  val eq : t -> t -> bool (* égalité *)  
  val order : t -> t -> bool (*  $\sqsubseteq^\sharp$  *)  
  val join : t -> t -> t (*  $x^\sharp, y^\sharp \mapsto \bigsqcup \{x^\sharp, y^\sharp\}$  *)  
  val meet : t -> t -> t (*  $x^\sharp, y^\sharp \mapsto \bigsqcap \{x^\sharp, y^\sharp\}$  *)  
  val top : t (*  $\top$  *)  
  val bottom : t (*  $\perp$  *)  
  val to_string : t -> string (* fonction d'affichage *)  
end
```

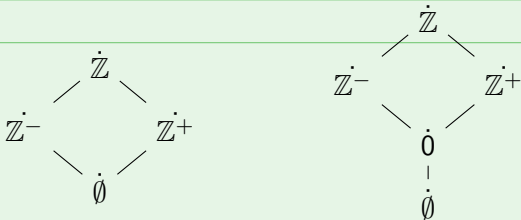
# Diagramme de Hasse

## Définition

**Diagramme de Hasse** : représentation graphique d'un ordre partiel  $\sqsubseteq$  sur un ensemble fini  $S$

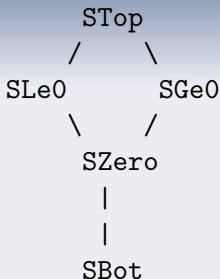
- les éléments de  $S$  sont représentés par des points
- si  $x \in S$  est inférieur à  $y \in S$  ( $x \sqsubseteq y$ ) alors on dessine le point  $x$  plus bas que  $y$
- on trace un trait entre  $x$  et  $y$  si et seulement si  $x$  est directement inférieur à  $y$  ( $x \sqsubseteq y$  et si  $x \sqsubseteq z \sqsubseteq y$  alors  $z = x$  ou  $z = y$ )

## Exemple



## ASCII art

```
<:finite_lattice<
```



→

```

SBot, SZero
| SZero, SLe0
| SZero, SGe0
| SLe0, STop
| SGe0, STop

```

```
>>
```

- Camlp4 nous donne la chaîne entre `<:finite_lattice<` et `>>`
- on lit le diagramme de Hasse ligne par ligne
- on génère le motif de filtrage de droite
- Camlp4 fait le remplacement dans le code source

# Génération du treillis

- on génère l'ordre par clôture réflexive transitive du résultat précédent
- borne supérieure (resp. inférieure) : pour chaque paire :
  - on calcule l'ensemble des majorants (resp. minorant)
  - on prend son minimum (resp. maximum) s'il existe
  - sinon, on n'est pas en présence d'un treillis complet
- $\top$  (resp.  $\perp$ ) : on prend le maximum (resp. minimum) de l'ensemble (s'il existe, sinon on n'est pas en présence d'un treillis complet)

## Propriété

Dans le cas d'ensembles  $S^\sharp$  finis, la donnée d'une borne supérieure ensembliste  $\bigsqcup^\sharp : \mathcal{P}(S^\sharp) \rightarrow S^\sharp$  est équivalente à la donnée de  $\perp$  et d'une borne supérieure binaire  $\sqcup^\sharp : S^\sharp \times S^\sharp \rightarrow S^\sharp$ .

## Exemple de code généré

```

type sSign2_base_t = | S0 | STop | SLe0 | SGe0 | SBot
module SSign2Lat : LatticeDef.Lattice with type t = sSign2_base_t = struct
  type t = sSign2_base_t
  let eq = ( = )
  let order a b = match (a, b) with
    | (S0, S0) | (S0, SGe0) | (S0, SLe0) | (S0, STop) | (SBot, S0) |
      (SBot, SBot) | (SBot, SGe0) | (SBot, SLe0) | (SBot, STop) |
      (SGe0, SGe0) | (SGe0, STop) | (SLe0, SLe0) | (SLe0, STop) |
      (STop, STop) -> true
    | _ -> false
  let join a b = match (a, b) with
    | (SBot, SBot) -> SBot
    | (S0, SGe0) | (SGe0, S0) | (SGe0, SGe0) | (SGe0, SBot) | (SBot, SGe0)
      -> SGe0
    | (S0, SLe0) | (SLe0, S0) | (SLe0, SLe0) | (SLe0, SBot) | (SBot, SLe0)
      -> SLe0
    | (S0, STop) | (STop, S0) | (STop, STop) | (STop, SLe0) | (STop, SGe0)
      | (STop, SBot) | (SLe0, STop) | (SLe0, SGe0) | (SGe0, STop) |
      (SGe0, SLe0) | (SBot, STop) -> STop
    | (S0, S0) | (S0, SBot) | (SBot, S0) -> S0
  let meet a b = [...]
  let top = STop
  let bottom = SBot
end

```

1 Interprétation abstraite

2 Construction du treillis

**3 Opérateurs abstraits**

4 Résultats



# Lecture de la fonction de concrétisation

Pour lire  $\gamma : S^\sharp \rightarrow \mathcal{P}(\mathbb{Z})$ , on utilise la grammaire Camlp4 :

```

EXTEND Gram
GLOBAL : str_item ;
str_item :
  [ [ "abstract_domain" ; id = UIDENT ;
      "on" ; order = patt ;
      "with" ; concr = gamma -> generate _loc id order concr ] ] ;
gamma :
  [ [ "gamma" ; v = LIDENT ; "=" ; "function" ; OPT "|" ;
      b = gamma_body -> _loc, v, b ] ] ;
gamma_body :
  [ [ c = gamma_case ; "|" ; l = SELF -> c : :l
      | c = gamma_case -> [c] ] ] ;
gamma_case :
  [ [ c = UIDENT ; "->" ; s = STRING -> _loc, c, s ] ] ;
END

```

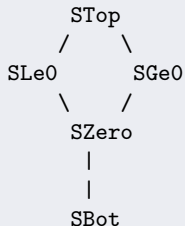
Pour chaque  $c$  de  $S^\sharp$  la chaîne de caractères  $s$  donne  $v \in \gamma(c)$  comme une formule SMT-LIB.

# Exemple

Spécification complète d'un domaine abstrait :

```
abstract_domain Sign on
```

```
<:finite_lattice<
```



```
>>
```

```
with gamma x = function
```

```
| STop -> "true"
| SLe0 -> "(<= x 0)"
| SGe0 -> "(>= x 0)"
| SZero -> "(= x 0)"
| SBot -> "false"
```

# Vérification de la fonction de concrétisation

$\gamma$  doit vérifier les propriétés :

- monotonie : si  $x^\# \sqsubseteq^\# y^\#$  alors  $\gamma(x^\#) \sqsubseteq \gamma(y^\#)$
- $\gamma(\top) = \mathbb{Z}$  (il existe toujours une abstraction correcte)
- correction de la borne inférieure binaire : pour tout  $x^\#$  et  $y^\#$  de  $S^\#$ ,  
 $\gamma(x^\#) \sqcap \gamma(y^\#) \sqsubseteq \gamma(x^\# \sqcap^\# y^\#)$

On teste cela avec un solveur SMT

# Signature du module OCaml

```

module type IntAbsDom = sig
  include Lattice

  val add : t -> t -> t (* addition en avant *)
  val sub : t -> t -> t (* soustraction en avant *)
  val mul : t -> t -> t (* multiplication en avant *)
  val add_bwd : t -> t -> t -> t * t (* addition en arrière *)
  val sub_bwd : t -> t -> t -> t * t (* soustraction en arrière *)
  val mul_bwd : t -> t -> t -> t * t (* multiplication en arrière *)
end

```

Une opération arrière raffine deux opérandes  $x^\sharp$  et  $y^\sharp$  lorsque  $z^\sharp$  sur-approxime le résultat (ex.  $\text{add\_bwd } \mathbb{Z}^- \ \mathbb{Z}^- \ \dot{0} = \dot{0}$ ),

# Sémantique avant

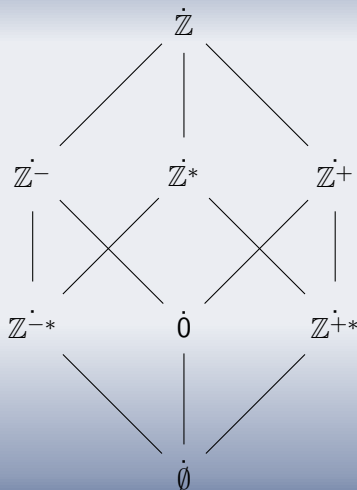
L'opérateur binaire  $binop^\sharp$  est une abstraction correcte de  $binop$  si :

$$\forall x^\sharp, y^\sharp \in S^\sharp, \forall x, y, z \in S, x \in \gamma(x^\sharp) \wedge y \in \gamma(y^\sharp) \wedge z = x \text{ binop } y \Rightarrow z \in \gamma(x^\sharp \text{ binop}^\sharp y^\sharp)$$

D'où la méthode de calcul de  $x^\sharp \text{ binop}^\sharp y^\sharp$  :

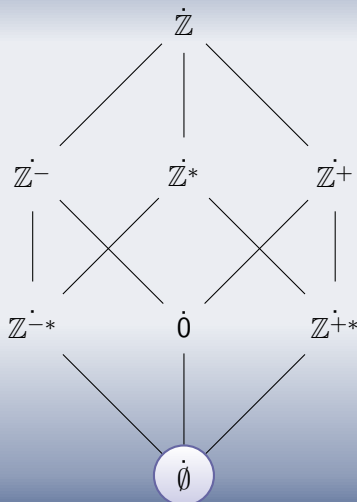
- pour tout  $z^\sharp$ , vérifier la formule précédente avec un solveur SMT ;
- choisir le plus petit des  $z^\sharp$  ayant passé le test précédent.

# Sémantique avant, exemple

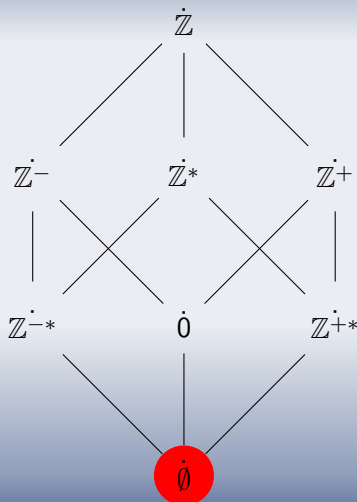
$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


## Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

$$\forall x, y. x > 0 \wedge y \geq 0 \Rightarrow x + y \in \emptyset$$


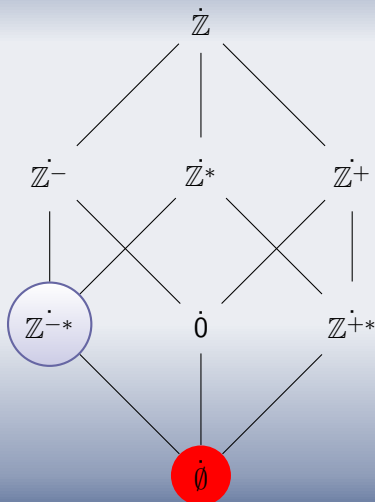
# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


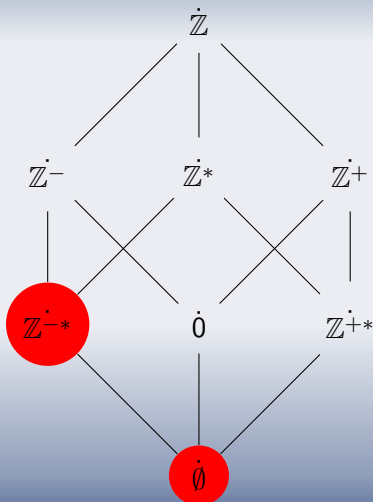


# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

$$\forall x, y. x > 0 \wedge y \geq 0 \Rightarrow x + y < 0$$


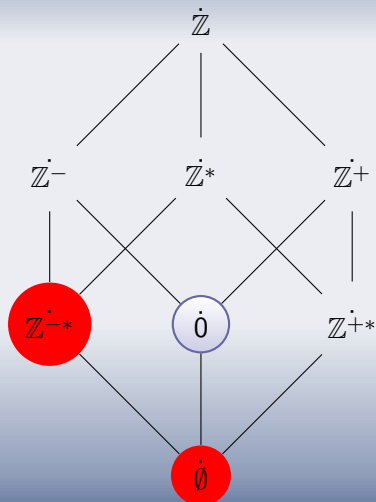
# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


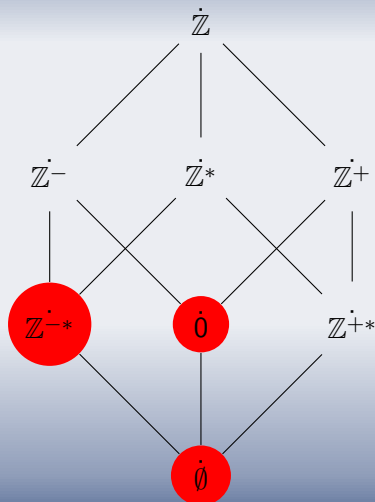
## Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

$$\forall x, y. x > 0 \wedge y \geq 0 \Rightarrow x + y = 0$$

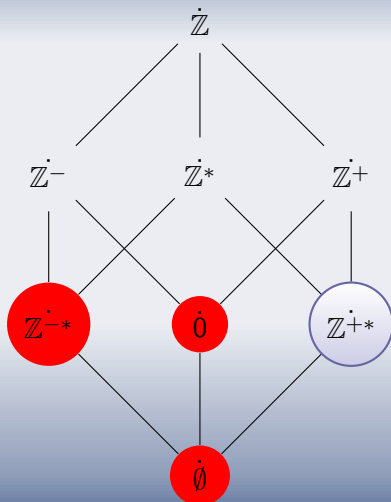


# Sémantique avant, exemple

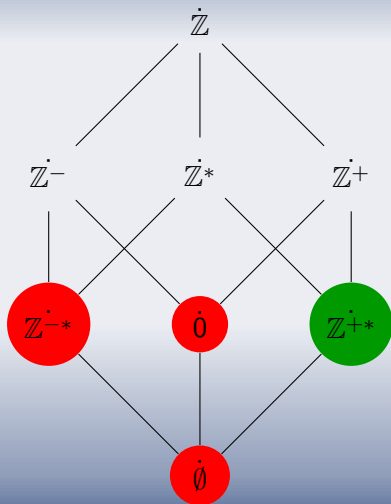
$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


## Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

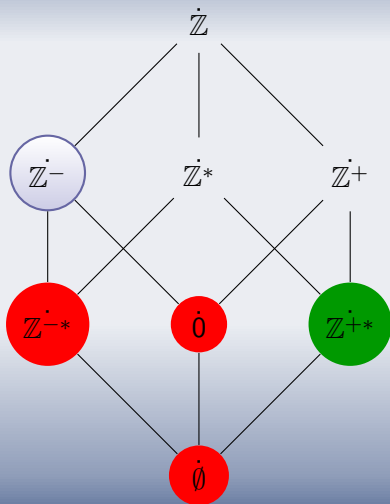
$$\forall x, y. x > 0 \wedge y \geq 0 \Rightarrow x + y > 0$$


# Sémantique avant, exemple

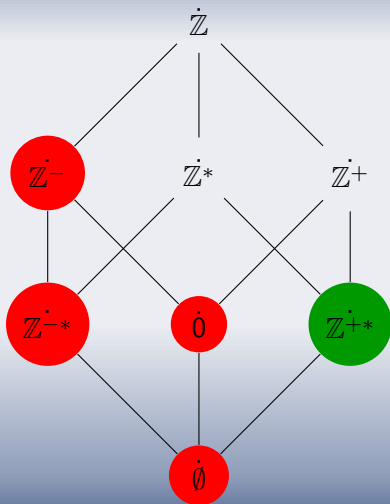
$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

$$\forall x, y. x > 0 \wedge y \geq 0 \Rightarrow x + y \leq 0$$


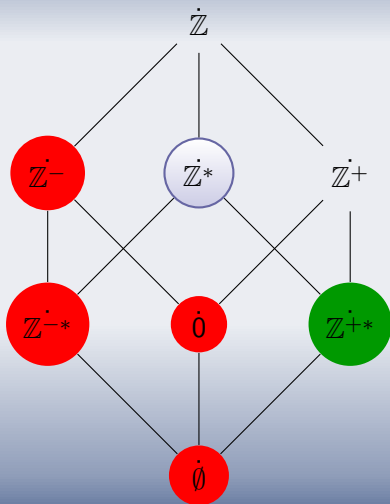
# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


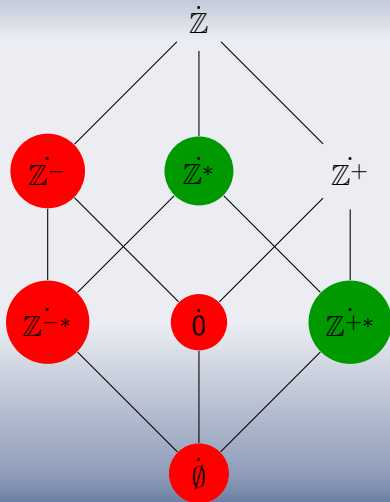


# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

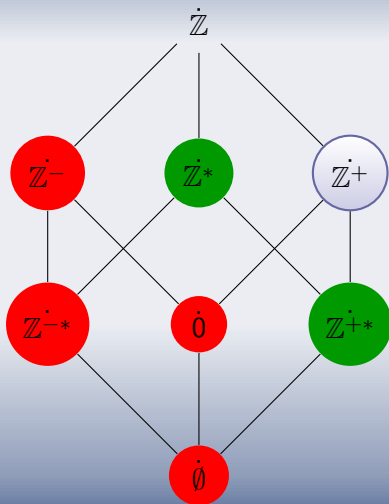
$$\forall x, y. x > 0 \wedge y \geq 0 \Rightarrow x + y \neq 0$$


# Sémantique avant, exemple

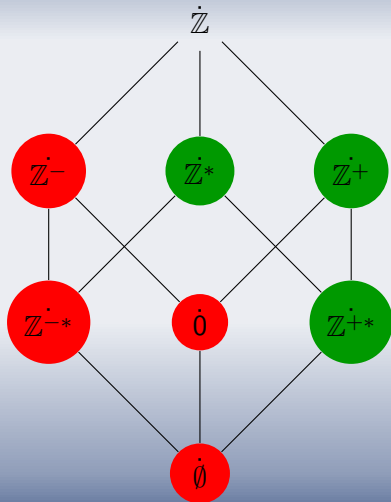
$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

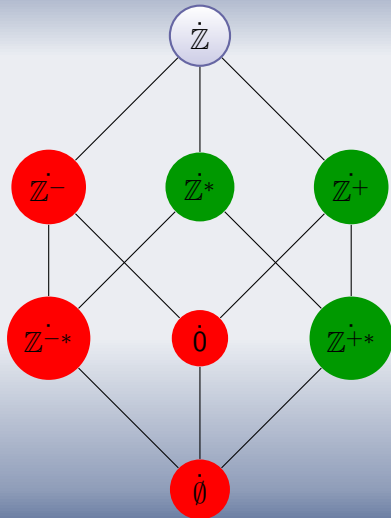
$$\forall x, y. x > 0 \wedge y \geq 0 \Rightarrow x + y \geq 0$$


# Sémantique avant, exemple

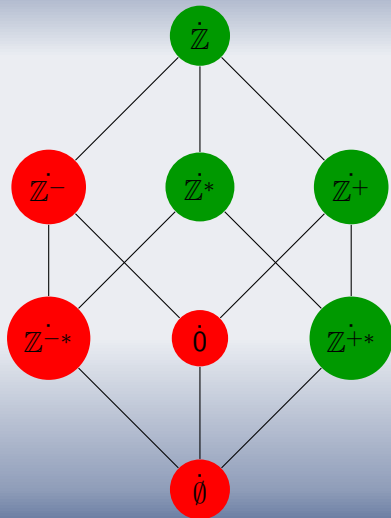
$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

$$\forall x, y. x > 0 \wedge y \geq 0 \Rightarrow x + y \in \mathbb{Z}$$


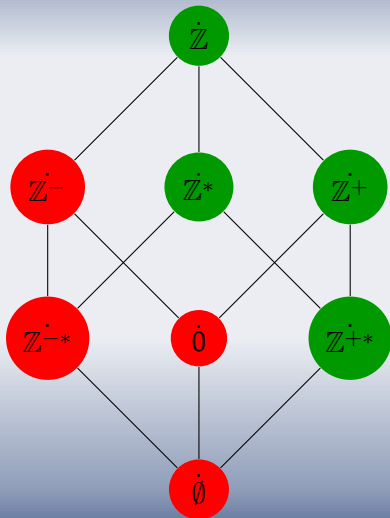
# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$


# Sémantique avant, exemple

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} :$$

$$\mathbb{Z}^{+*} +^{\#} \mathbb{Z}^{+} = \mathbb{Z}^{+*}$$



# Sémantique arrière

Formule différente mais méthode similaire :

$$\begin{aligned} \forall x^\#, y^\#, z^\#, x'^\#, y'^\# \in S^\#, \forall x, y, z \in S, x \in \gamma(x^\#) \wedge y \in \gamma(y^\#) \wedge \\ z = x \text{ binop } y \wedge z \in \gamma(z^\#) \wedge \\ (x'^\#, y'^\#) = \text{binop}^\# \downarrow (x^\#, y^\#, z^\#) \Rightarrow \\ x \in \gamma(x'^\#) \wedge y \in \gamma(y'^\#) \end{aligned}$$

On remarque qu'on peut calculer  $x'^\#$  et  $y'^\#$  séparément.



# Optimisations

Pour limiter le nombre d'appels aux solveurs, on peut exploiter les propriétés suivantes :

- monotonie de  $binop^\sharp$  ;
- lorsque un  $z^\sharp$  est une abstraction correcte, la meilleure abstraction est un élément plus petit ;
- lorsque un  $z^\sharp$  n'est pas une abstraction correcte, les éléments plus petit ne le sont pas non plus.

1 Interprétation abstraite

2 Construction du treillis

3 Opérateurs abstraits

**4 Résultats**

## Sur quelques exemples classiques

Domaine	taille de $S^\#$	sans optimisation (temps total)	avec optimisations (temps total)
bot_top	2	49 (1 s)	40 (1 s)
eq_zero	4	628 (6 s)	339 (4 s)
signs	4	628 (7 s)	300 (3 s)
signs_with_zero	5	1540 (17 s)	594 (7 s)
extended_signs	8	7500 (83 s)	2429 (33 s)
parity	4	617 (8 s)	325 (4 s)
mod3	5	1294 (147 s)	600 (117 s)

Nombre d'appels aux solveurs SMT et temps d'exécution  
(sur un Core 2 @ 1.20 GHz)

## Quelques remarques

- + code compact (un millier de lignes de Caml)
- + on obtient la meilleure abstraction sur l'essentiel des exemples
- - on est limité à des domaines finis (sans grand usage pratique)...
- - ...de petite taille  
(complexité en  $\Omega(n^3)$  pour un treillis de  $n$  éléments)
- - difficile de prouver certaines propriétés de congruence  
ou faisant intervenir la multiplication avec les solveurs utilisés

# Questions

?