Sémantique et analyse de programmes analyse statique par interprétation abstraite

David Monniaux

CNRS / VERIMAG

Mardi 6 avril 2010

Plan

Abstraction booléenne

On remplace un état « compliqué » — couple (point de programme, valeur des variables) par un état d'un automate fini.

Les états de l'automate fini sont des couples (point de programme, formule sur des prédicats)

On place une flèche entre deux états a, b de l'automate fini s'il y a une façon de passer d'un état représenté par b en un pas de calcul.

Tester cela est un problème SMT, coûteux à décider.

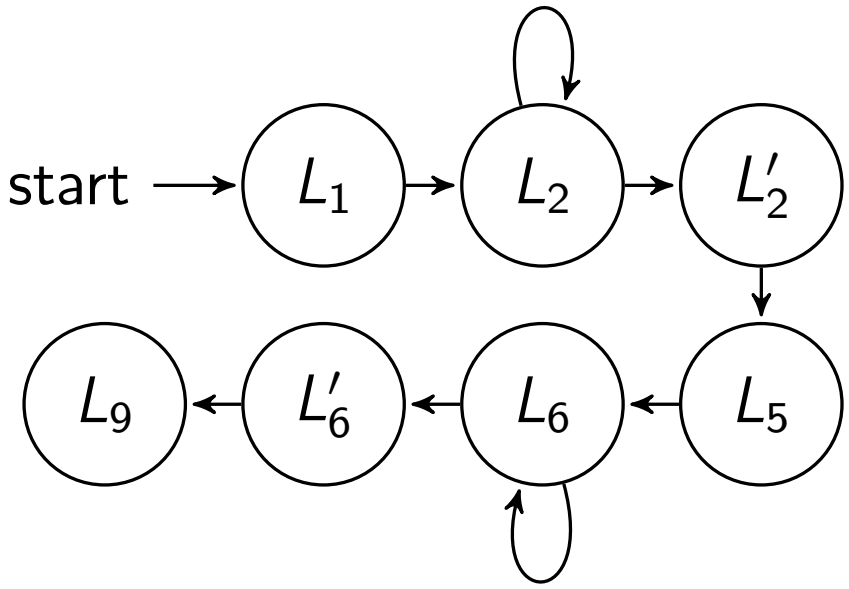
Exemple

```
x = 0; /* ligne 1 */
while /* ligne 2 */ (x < 10) {
   x = x+1;
}
y = 0; /* ligne 5 */
while /* ligne 6 */ (y < x) {
   y = y+1;
}
/* ligne 9 */
```

On essaye les prédicats $x < 0$, $x = 0$, $x > 0$, $x < 10$, $x = 10$, $x > 10$, $y < 0$, $y = 0$, $y > 0$, $y < x$, $y = x$, $y > x$.

Note : 12 prédicats, dans le pire cas $2^{12} = 4096$ combinaisons, dont certaines sont impossibles (on ne peut pas avoir $x < 0$ et $x > 0$ en même temps).

Automate abstrait




```
x = 0; /*L1*/
while /*L2*/ (x<10)
   x = x+1;
}
y = 0; /*L5*/
while /*L6*/ (y<x) {
   y = y+1;
}
/*L9*/
```

L_1 : ligne 1, $x = 0$
L_2 : ligne 2, $0 < x < 10$
L'_2 : ligne 2 : $x = 10$
L_5 : ligne 5 : $x = 10$
L_6 : ligne 6 : $x = 10 \wedge y < x$
L'_6 : ligne 6 : $x = 10 \wedge y = x$
L_9 : ligne 9 : $x = 10 \wedge y = x$

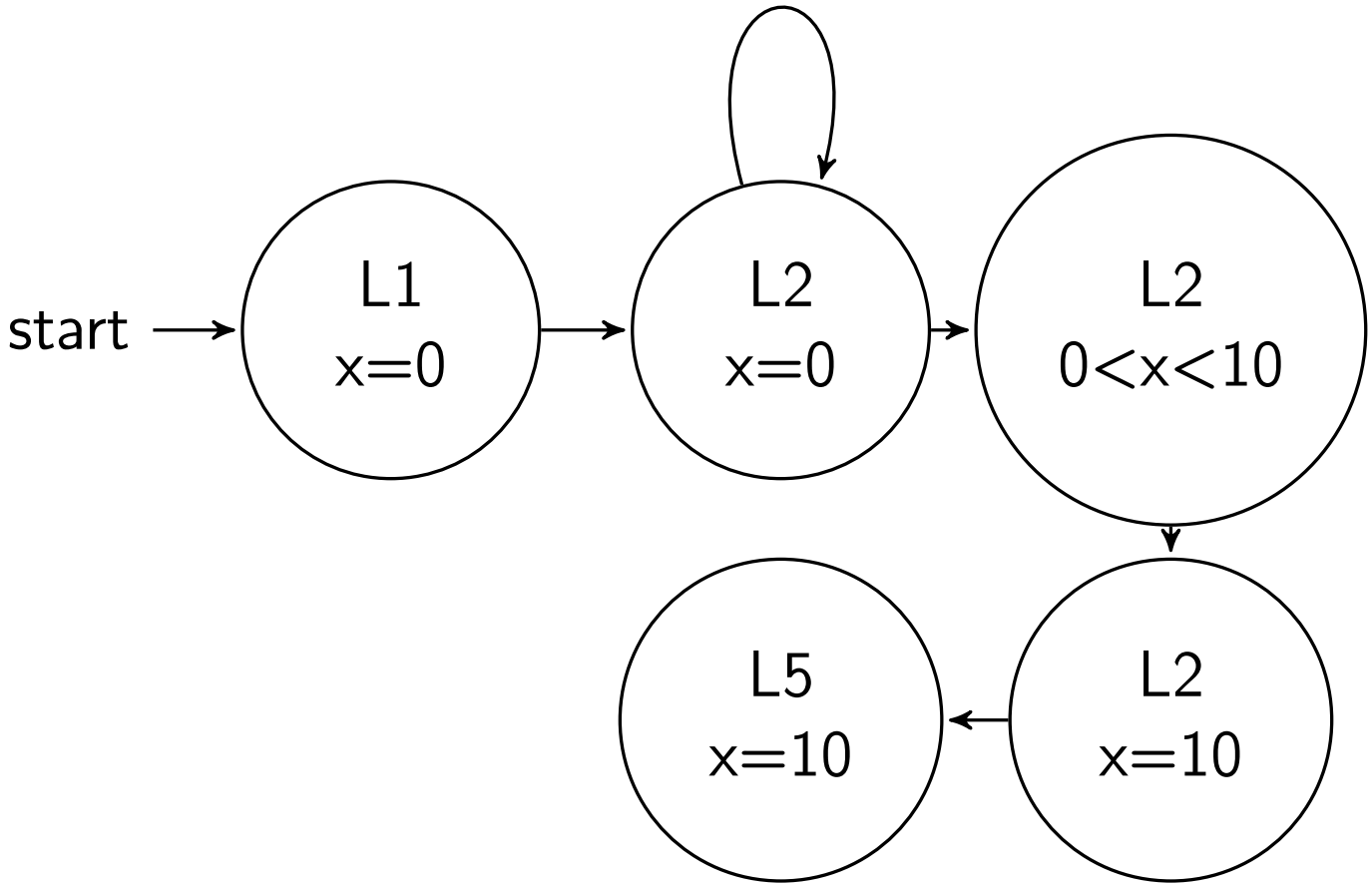
Attention

```
x = 0;
while (x != 10) { /* L2 */
   x = x+2;
}
/* L5 */
```

Et choix naïf (syntaxique) de prédicats ($x < 0$, $x = 0$, $x > 0$, $x < 10$, $x = 10$, $x > 10$).

Solution fausse ?

Certains ont répondu :



Pourquoi cette solution est fausse ?

Cette solution représente effectivement l'ensemble des comportements du programme... elle est donc **correcte**.

Mais vous ne rendez compte de cela que parce que vous avez déjà dans la tête l'ensemble des états accessibles ! **Vous trichez !**

Le problème est qu'elle n'est pas **inductive** : autrement dit, on peut passer d'un état effectivement représenté dans le graphe à un état qui ne l'est pas !

Attention

```
x = 0;
while (x != 10) { /* L2 */
   x = x+2;
}
/* L5 */
```

Si ligne 2, vous savez que vous pouvez être dans un état abstrait $0 < x < 10$, alors notamment $x = 9$ convient.

Oui, $x = 9$ est concrètement inaccessible ! Mais ça vous ne le savez que parce que vous supposez connu l'ensemble des états concrets accessibles $\{0, 2, 4, 6, 8\}$.

Et donc, normalement, il faudrait une transition de $0 < x < 10$ ($x = 9$) à un nouvel état $x > 10$ ($x = 11$).

La grosse différence

L'élève de l'ENSIMAG est intelligent(e), il voit le programme et sait que l'ensemble des accessibles est $\{0, 2, 4, 6, 8\}$ parce que x est forcément pair.

Il/elle projette sur les prédicats et trouve que $x > 10$ est inaccessible.

La machine est bête **ne "voit" pas que c'est forcément pair** parce qu'on ne lui a pas demandé de regarder le prédicat $x \bmod 2 = 0$.

On demande un procédé **algorithmique** d'analyse.
« On voit bien que » ne s'automatise pas.

Pas convaincu ?

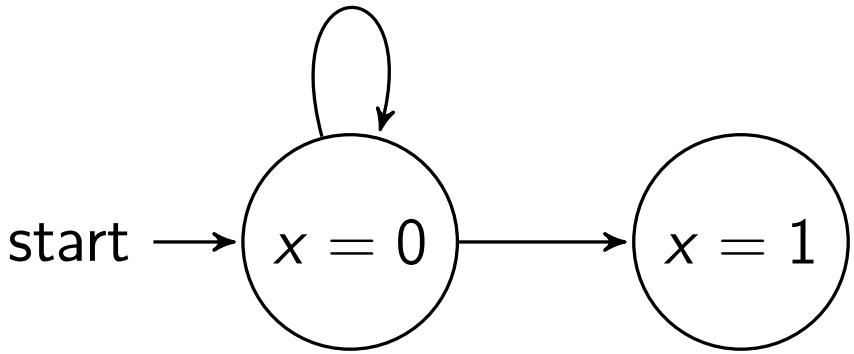
Soit P un programme où n'intervient jamais la variable booléenne x. Considérons : $x := 0\,;\ P\,;\ x := 1$

On se donne comme prédicats $x = 0$ et $x = 1$. Donner un automate minimum qui représente l'ensemble des comportements du programme par rapport à x...

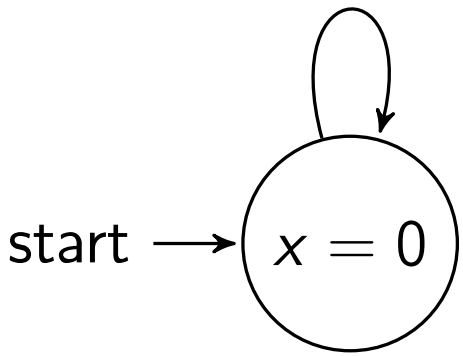
Automate à deux états $x = 0$, $x = 1$. Simple, non ?

L'automate minimal (non inductif)

Si P **termine** :



Si P **ne termine pas** :



En résumé

- Il ne faut pas espérer calculer la meilleure abstraction possible (= arrêt des machines de Turing, théorème de Rice etc.).
- On peut par contre calculer un abstraction valide si on connaît juste ce qu'il y a marqué dans les états et pas en raisonnant en supposant connu l'ensemble des accessibles.
- Cette abstraction peut être mauvaise (cf $x > 10$ accessible abstraitement alors qu'il ne l'est pas en vrai).
- Ce genre de problèmes est inévitable.
- On l'aura pour **toutes les méthodes d'analyse correctes**.

Plan

Rappel de l'idée

On va essayer de calculer un intervalle pour chaque point de programme et chaque variable, en faisant de l'**arithmétique d'intervalles** :

```
assume(x >= 0 && x<= 1);
assume(y >= 2 && y= 3);
assume(z >= 3 && z= 4);
t = (x+y) * z;
```

Intervalle de z ?

Rappel de l'idée

On va essayer de calculer un intervalle pour chaque point de programme et chaque variable, en faisant de l'**arithmétique d'intervalles** :

```
assume(x >= 0 && x<= 1);
assume(y >= 2 && y= 3);
assume(z >= 3 && z= 4);
t = (x+y) * z;
```

Intervalle de z ? $[6, 16]$

Pourquoi c'est intéressant ?

Soit $t(0..10)$ un tableau dans un programme.
Le programme écrit $t(i)$.

Il faut savoir si $0 \leq i \leq 10$, autrement dit connaître un **intervalle** sur i.

De nouveau...

```
assume(x >= 0 && x <= 1);
y = x;
z = x-y;
```

- L'élève de l'ENSIMAG (intelligent) voit $z = 0$ car il tient compte de $y = x$.
- L'arithmétique d'intervalle ne le voit pas car elle ne tient pas compte de $y = x$. Si on veut tenir compte de $y = x$ il faut avoir prévu dans l'algorithme de pister ces relations.

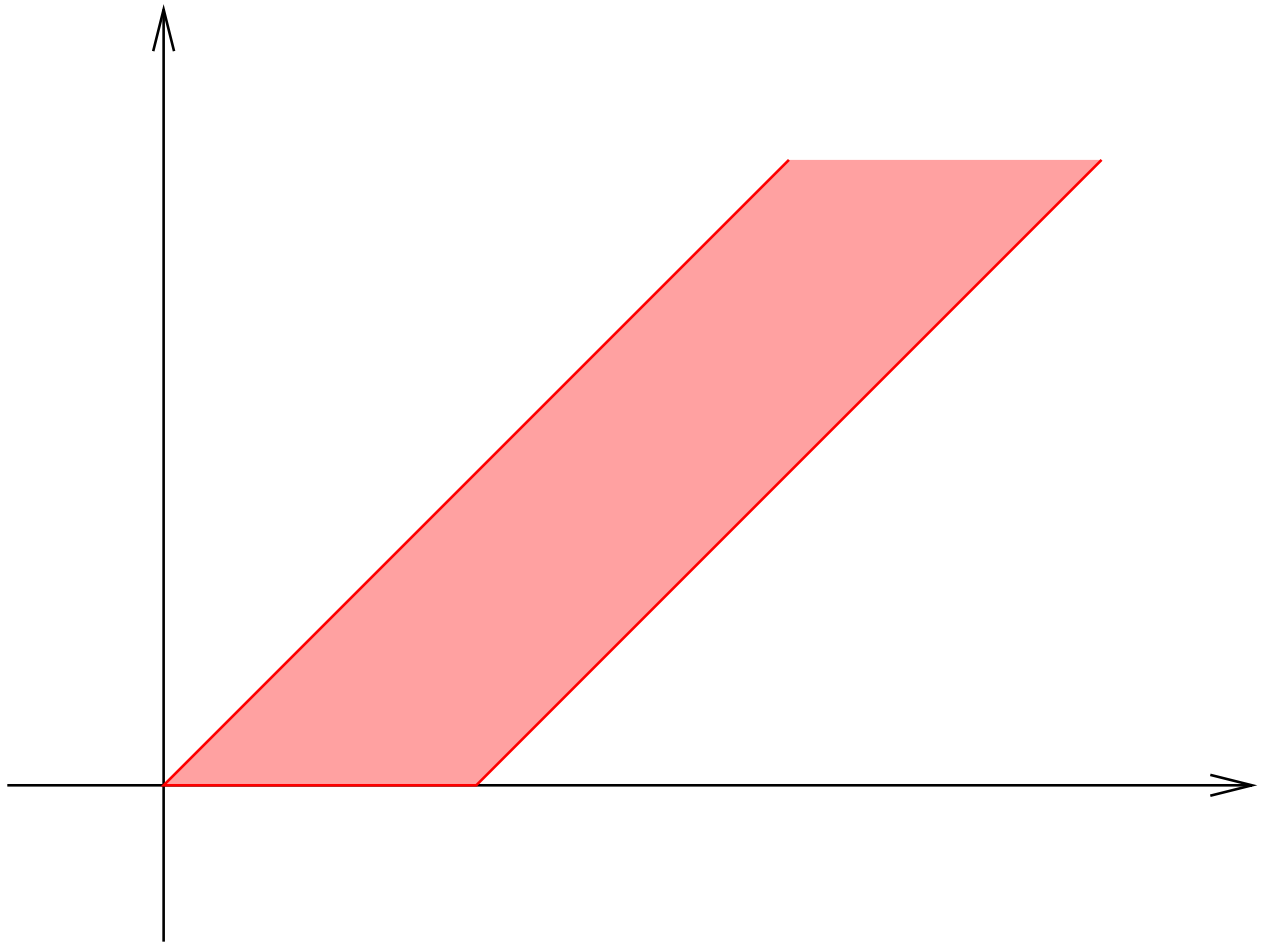
Comment pister les relations

En utilisant des **domaines relationnels**.

Par exemple : garder pour chaque variable un intervalle, et pour chaque paire de variables (x, y) une information de la forme $x - y \leq C$. (On peut obtenir $x = y$ par $x - y \leq 0$ et $y - x \leq 0$.)

Mais comment **calculer** là-dessus ?

Bornes de différences



Exemple pratique

Si on sait que $x - y \leq 4$ et que l'on calcule $z = x + 3$, alors on sait que $z - y \leq 7$.

Si l'on sait que $x - z \leq 20$, que $x - y \leq 4$ et que $y - z \leq 6$, alors on sait que $x - z \leq 10$. (Cf plus court chemin.)

Plus généralement, on sait calculer sur ce type de relations.
Et sur notre exemple, on aurait conclu $z = 0$.

Pourquoi c'est utile

Soit $t(0..\mathbf{n})$ un tableau dans un programme.
Le programme écrit $t(i)$.

Il faut savoir si $0 \leq i \leq n$, autrement dit connaître des bornes sur i et sur $n - i$...

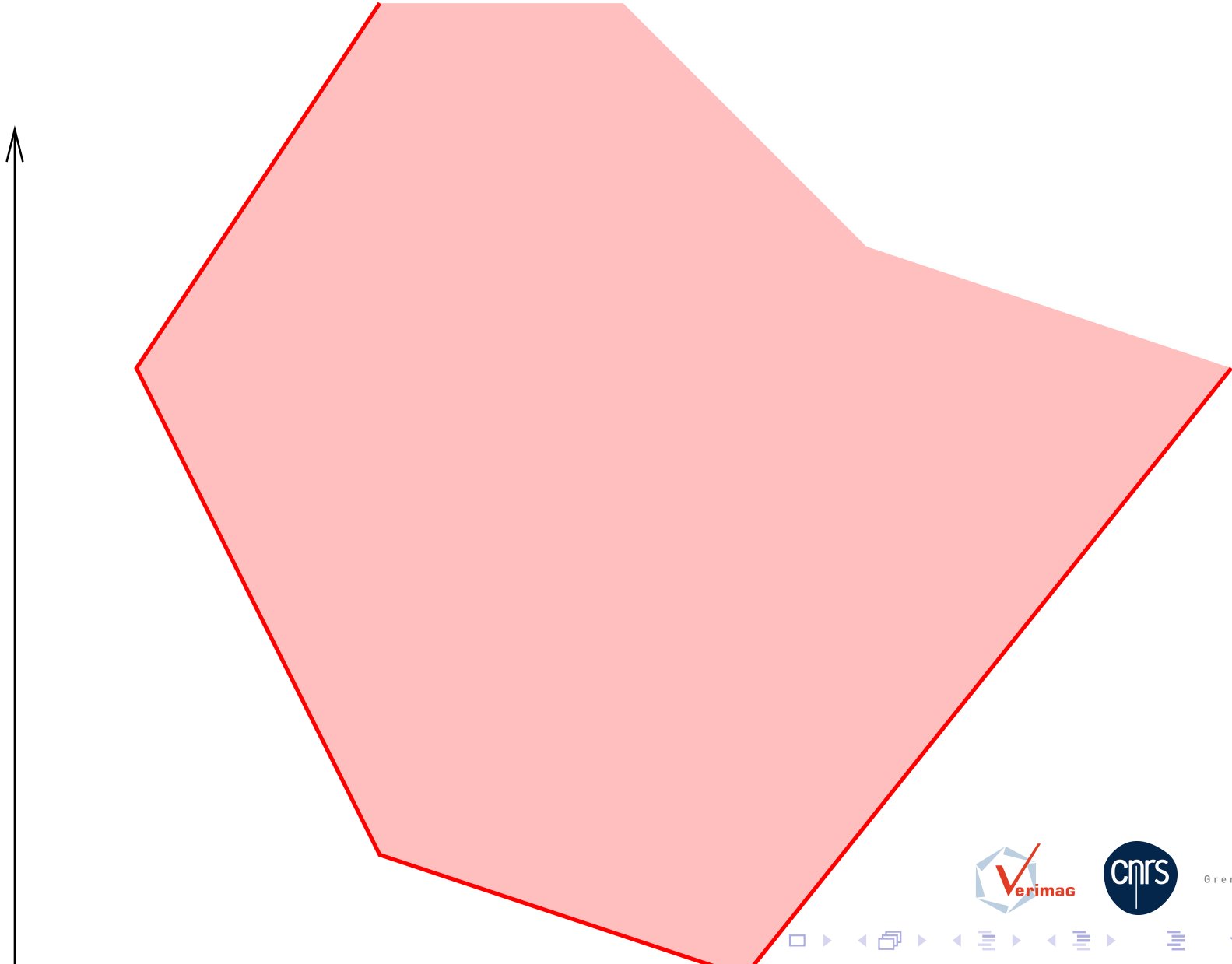
On peut faire mieux

Et si on pistait des relations comme $2x + 3y \leq 6$?

En un point de programme, on va avoir une conjonction d'inégalités linéaires.

Autrement dit, un **polyèdre convexe**.

Exemple de polyèdre



Mais attention

Plus on est précis, plus cela peut coûter cher. Pour chaque ligne de code :

- Intervalles : algorithmes $O(n)$, n nombre de variables.
- Différences $x - y \leq C$: algorithmes $O(n^3)$, n nombre de variables.
- Polyèdres : algorithmes souvent $O(2^n)$.

Sur des exemples de trois lignes et trois variables, tout fonctionne bien... mais en général ?

Encore plus fort

```
y = abs(x);   /* valeur absolue */
if (y >= 1) {
   assert(x != 0);
}
```

Encore plus fort

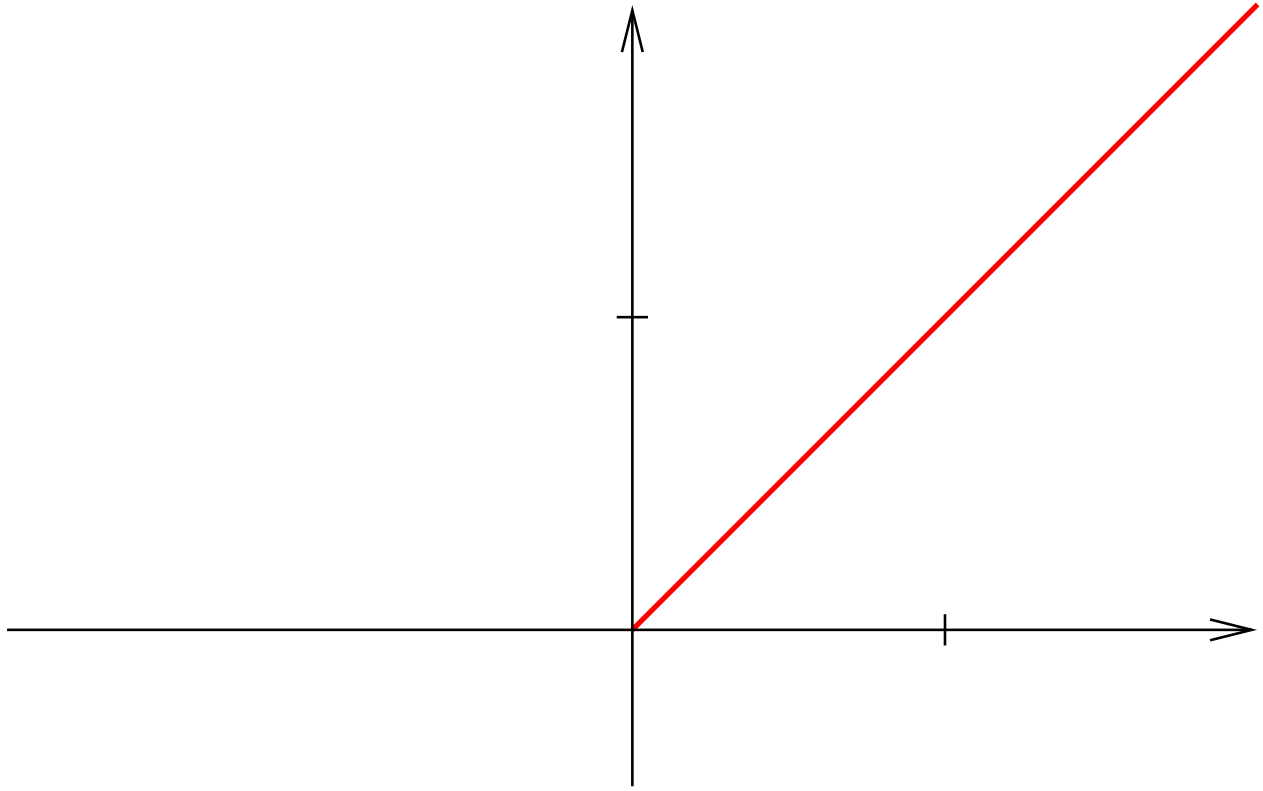
Intervalles :

```
/* -1000 <= x <= 2000 */
if (x < 0) y = -x; /* 0 <= y <= 1000 */
else y = x; /* 0 <= y <= 2000 */

if (y >= 1) { /* 1 <= y <= 2000 */
  assert(x != 0); /* -1000 <= x <= 2000 !!! */
}
```

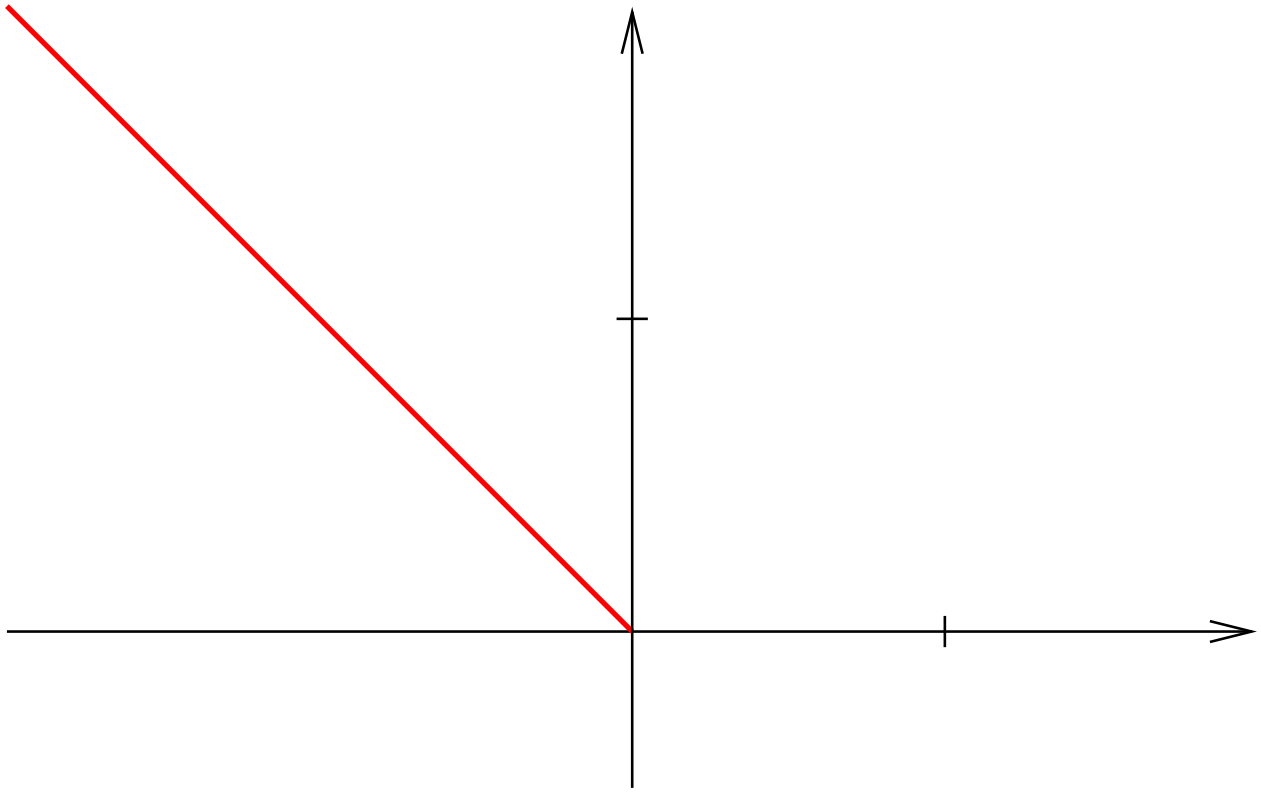
Et en polyèdres

Branche $x \geq 0$



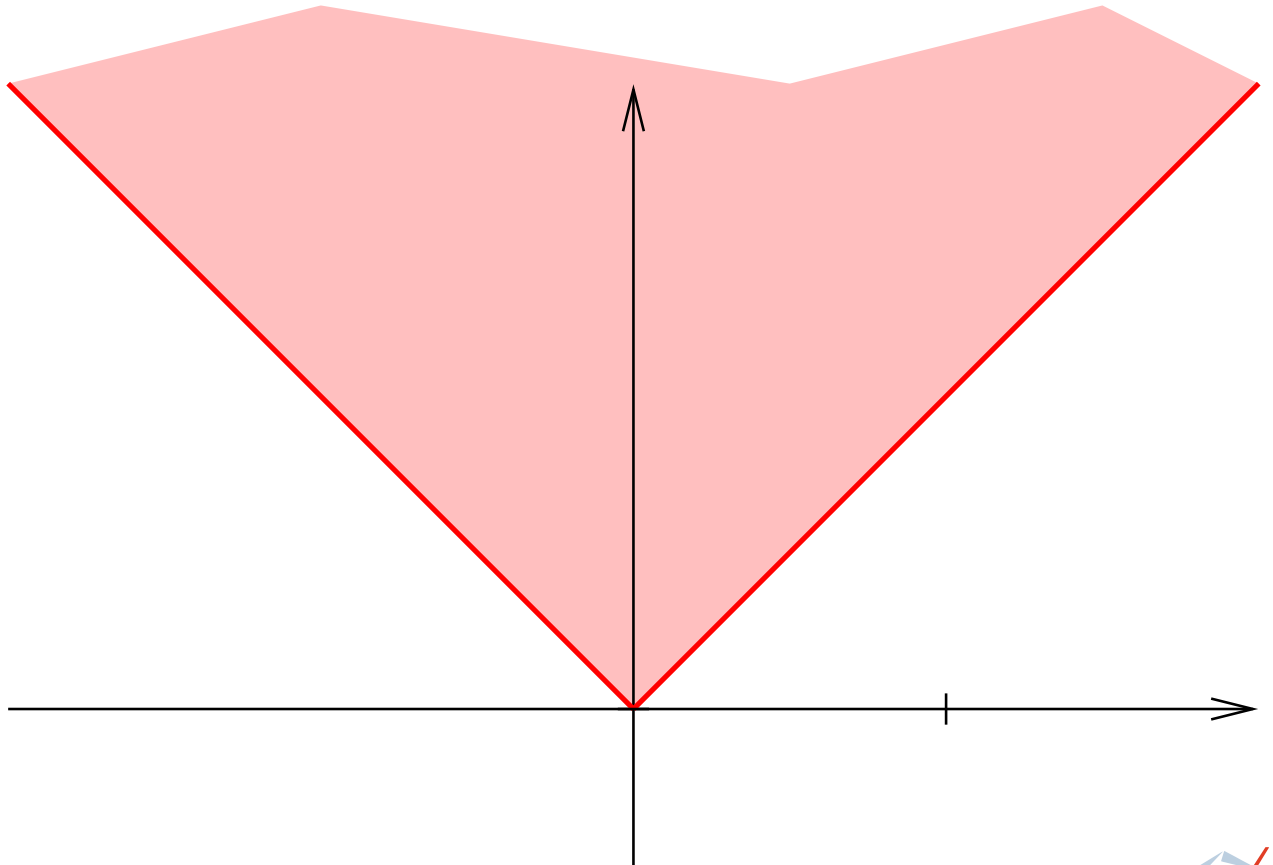
Autre branche du test

Branche $x < 0$



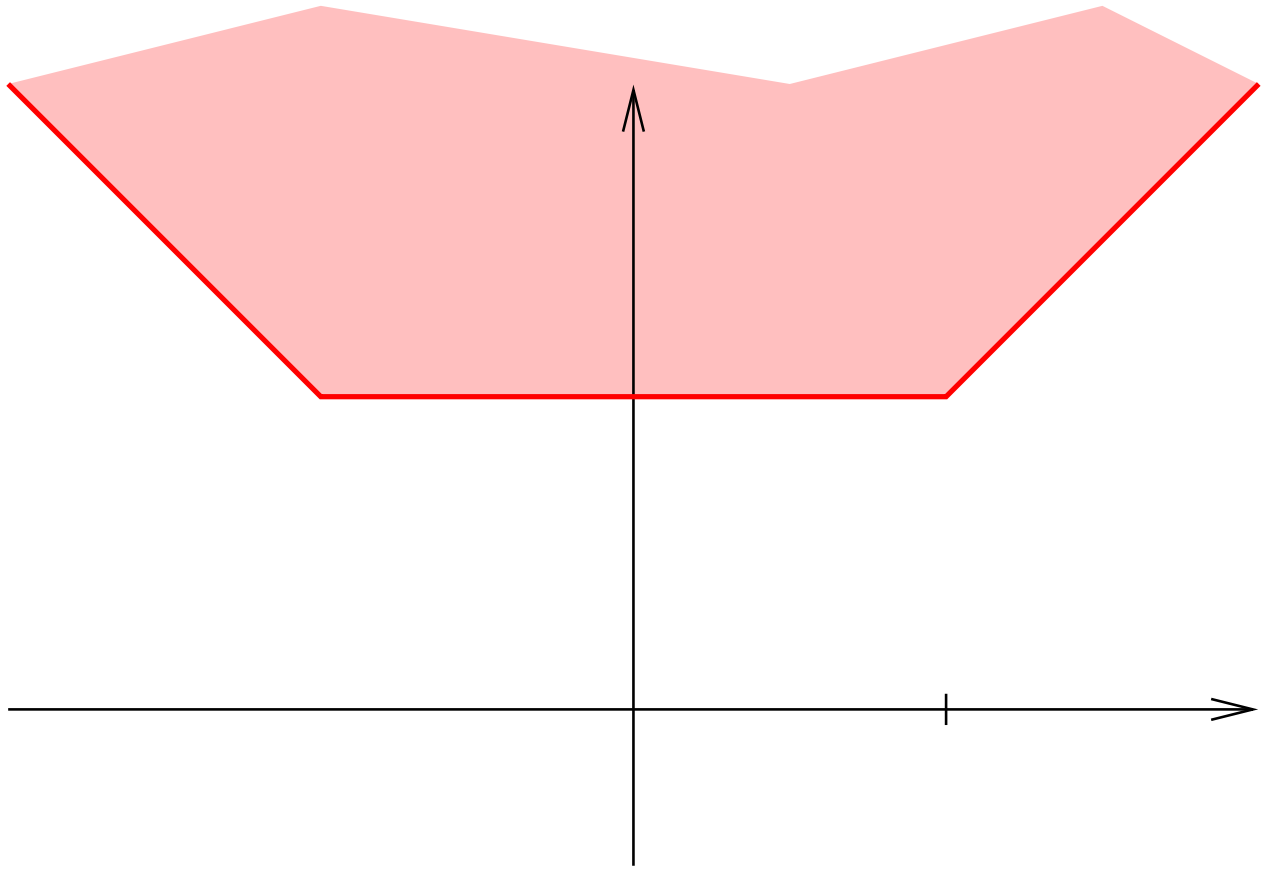
Après le premier test

$y = |x|$ n'est est l'union des deux lignes rouges. Ce n'est pas un convexe, on doit prendre le plus petit polyèdre qui le contient (en rose) — enveloppe convexe des deux lignes.



Dans le second test

Notez : contient $(x, y) = (0, 1)$.



Disjonctage

On aurait pu s'en sortir en écrivant une union de deux polyèdres :

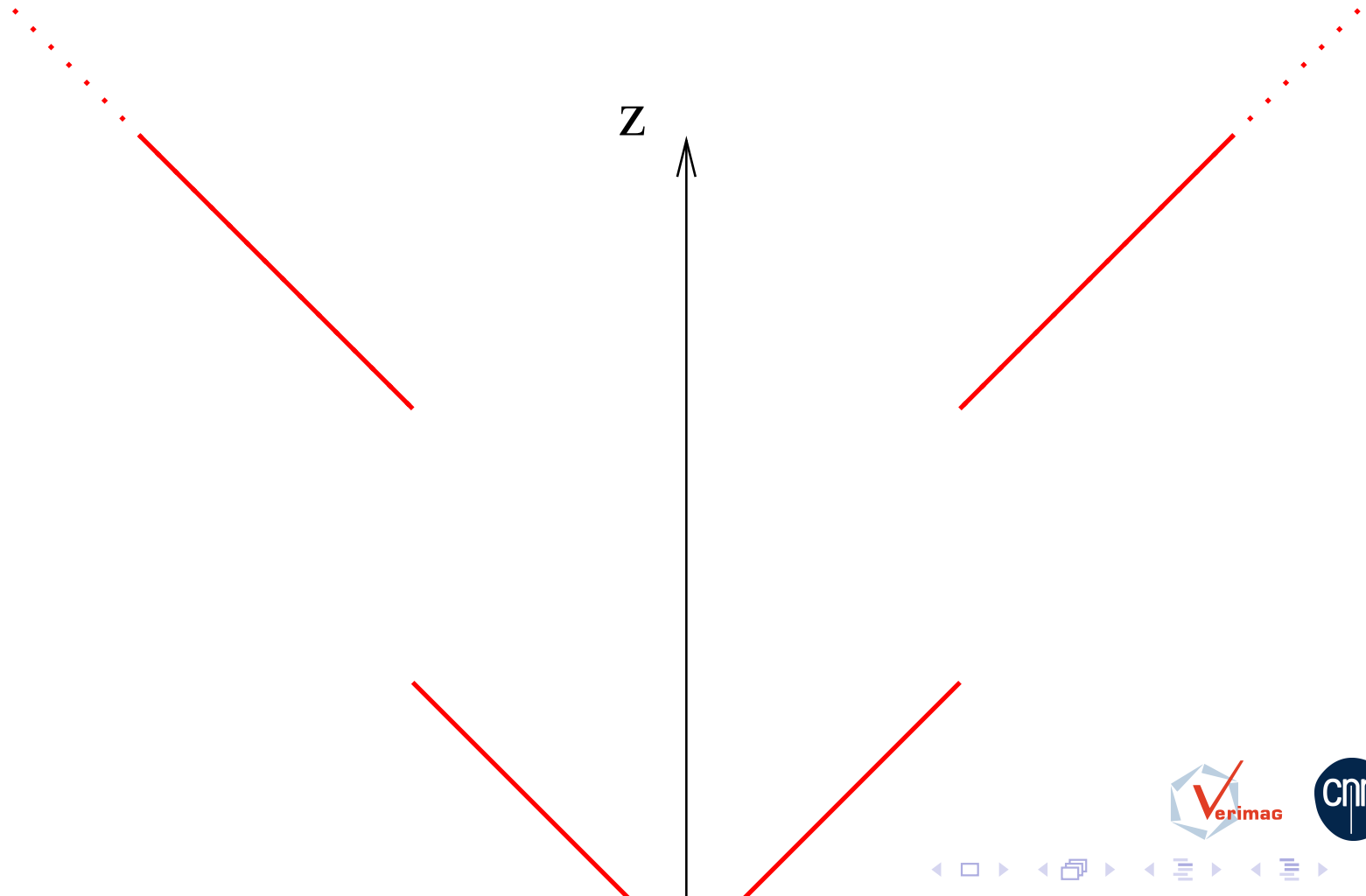
- $x \geq 0 \wedge y = x$
- $x < 0 \wedge y = -x$

Mais si on fait n tests ?

Deux tests

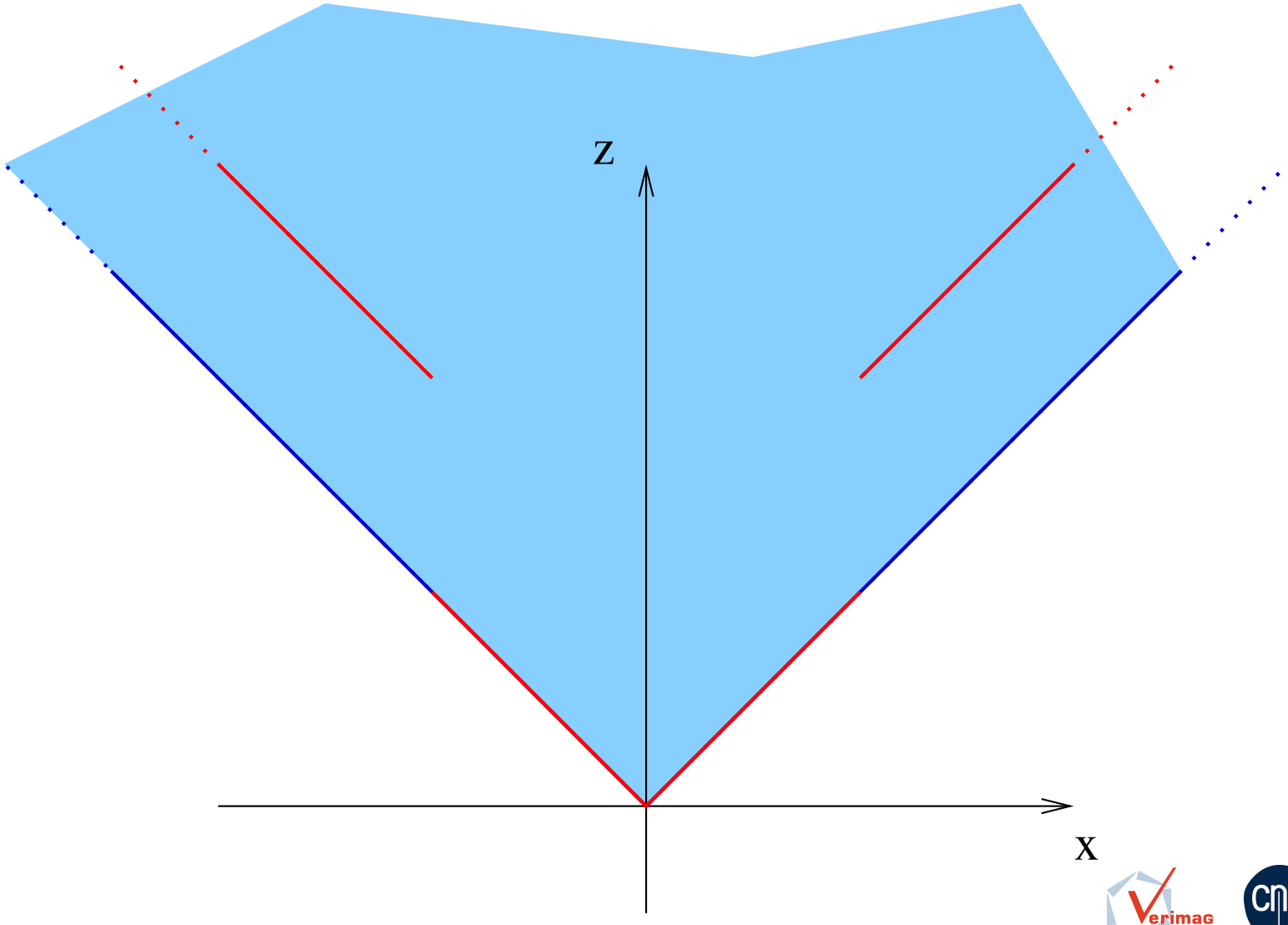
```
if (x >= 0) y=x; else y= -x;
if (y >= 1) z=y+1; else z=y;
```

4 polyèdres = calculs coûteux



Deux tests, enveloppe convexe

Beaucoup plus imprécis :



Source d'imprécision

- Il faudrait distinguer chaque chemin dans les tests et au bout on aurait un polyèdre.
- Mais il y a 2^n chemins.
- C'est **trop coûteux** (du moins sans méga-astuce).
- Dans les vrais outils, on ne distingue pas tous les chemins.
- **Explique certaines imprécisions.**

Plan

Et les boucles ?

On pousse les intervalles / polyèdres en avant...

```
int x=0;
while (x<1000) {
   x=x+1;
}
```

On tourne dans la boucle, on calcule $[0, 0]$, $[0, 1]$, $[0, 2]$, $[0, 3]$,... Comment ? $\phi(X) = \text{état initial} \sqcup \text{post}(X)$, donc

$\phi([a, b]) = \{0\} \sqcup [a + 1, \min(b, 999) + 1]$ (ce qui se passe sur les intervalles)x

Mais quand est-ce qu'on s'arrête ? Attendre 1000 tours ? Nooon !

Une seule solution...

L'**extrapolation** !

$[0,0]$, $[0,1]$, $[0,2]$, $[0,3] \rightarrow [0,+\infty)$

On recommence à pousser l'intervalle :

```
int x=0; /* [0, 0] */
while /* [0, +infty) (x<1000) {
   /* [0, 999] */
   x=x+1;
   /* [1, 1000] */
}
```

Ouf ! $[0,\infty[$ est stable !

Un résultat médiocre

On attendait $[0, 999]$. On a eu $[0, +\infty)$.

On fait tourner un tour la boucle :

```
[0, +infty) (x<1000)
  /* [0, 999] */
  x=x+1;
  /* [1, 1000] */
```

On obtient $\{0\} \sqcup [1, 1000] = [0, 1000]$.

Rétrécissement

```
int x=0; /* [0, 0] */
while /* [0,1000] (x<1000) {
   /* [0, 999] */
   x=x+1;
   /* [1, 1000] */
}
```

Ouf : $[0, 1000]$ est un invariant inductif.

Stabilisation

On cherche un ensemble (polyèdre, intervalles)

- Qui contienne les valeurs de départ de la boucle.
- Tel que si on propage dans un tour de boucle on reste dedans.

Autrement dit, on cherche X tel que $\phi(X) \subseteq X$ avec $\phi(X) = \text{états initiaux} \cup \text{post}(X)$ avec post(X) = états obtenus à partir de X en un tour de boucle.

C'est à dire **un** invariant inductif. (Pas forcément le plus petit.)

Et comment obtenir cela ?

On ne sait/veut pas calculer en général post(P) avec P intervalle (le corps de boucle peut être compliqué, avec des tests etc.).

On remplace le calcul par un calcul « surapproximant » mais plus simple post(X) $\subseteq$ post$^\sharp(X)$.

Et on ne sait pas faire $\cup$ sur les polyèdres, on fait $\sqcup$ (enveloppe convexe), donc finalement on calcule
$\phi^\sharp(X) = \text{états initiaux} \sqcup \text{post}^\sharp(X)$
au lieu de $\phi(X) \subseteq X$ avec $\phi(X) = \text{états initiaux} \cup \text{post}(X)$

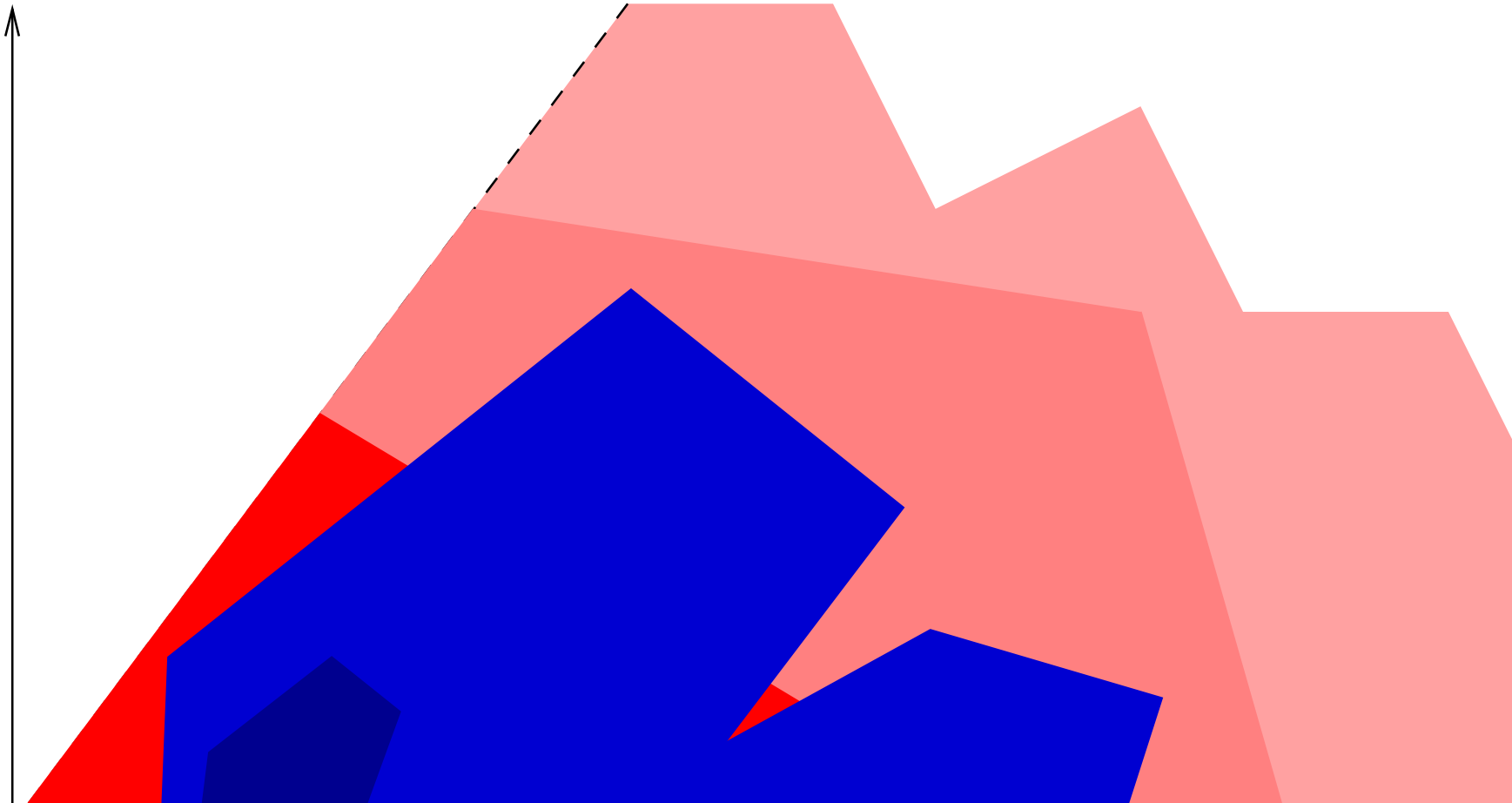
On surapproxime tout le temps

$\phi(X) \subseteq \phi^\sharp(X)$ donc lfp $\phi \subseteq$ lfp $\phi^\sharp$ (exercice de TD).

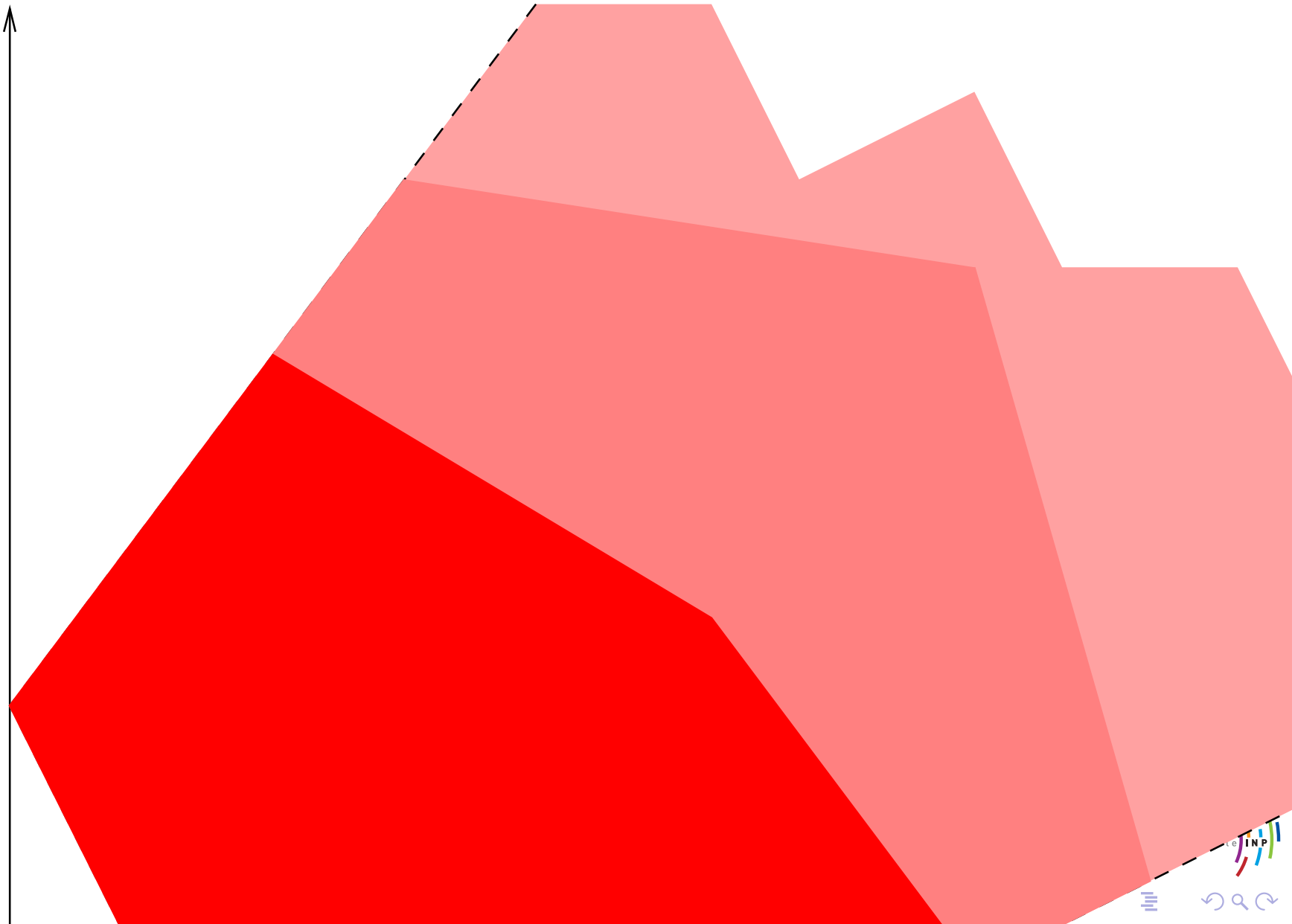
Donc au final on obtient une **surapproximation** du plus petit point fixe de ϕ.

Vision graphique

Bleu foncé = vrais états après 1 tour de boucle
Bleu plus clair = vrais états après 2 tours de boucle
Rouge foncé = états surapproximés après 1 tour de boucle
Rouge plus clair = états surapproximés après 2 tours de boucle



Extrapolation



Moralité

- On surapproxime pendant le calcul (même sans boucles).
- On surapproxime à l'extrapolation (élargissement).
- On obtient donc forcément un **sur**-ensemble des états accessibles.
- Ce sur-ensemble est un invariant inductif (on ne peut pas en sortir en exécutant la boucle).

Conséquence pratique

- On ne peut jamais prouver qu'un problème arrive vraiment (on ne sait pas si c'est juste dans la surapproximation ou dans la réalité).
  Exemple : intervalle trop grand $i \in [0, 20]$ pour accès $t(0..10)$, est-ce que l'intervalle est exact ?
- On est juste sûr que l'entièreté des problèmes potentiels est signalée (parce qu'on surapproxime les problèmes).
- Dans PolySpace, les avertissements colorient le code en orange.

Plan

Et en dehors des numériques

On peut analyser pointeurs, tableaux, mémoire, threads...

Mais ça devient encore plus compliqué à expliquer !

Points importants

- L'ordinateur est stupide, il ne « voit » pas pourquoi un programme fonctionne.
- C'est normal, tout ce qui est important est **indécidable algorithmiquement** (ou de **complexité élevée**).
- On cherche des invariants inductifs que l'on peut **prouver automatiquement** (p.ex. par propagation d'intervalles, de polyèdres pour les quantités numériques).
- Ils surapproximent les comportements, les problèmes.

Success stories

- **Microsoft SLAM** / Device driver verifier — predicate abstraction, contrôle le respect de l'API Windows dans des pilotes
- **PolySpace Verifier**, domaines numériques
- **Astrée**, domaine numériques spécialisés contrôle commande — projets A340, A380 (Airbus), ATV (EADS Astrium / ESA)
- **Absint**, analyse de temps d'exécution dans le pire cas en présence de pipelines et de cache