# Analyses arrières

- Debugging abstrait, slicing;
- sémantique abstraite arrière;
- itérations décroissantes locales;
- combinaison d'analyses avant et arrière.

# Un cas simple : le slicing

Après avoir exécuté un programme, on trouve une valeur erronée pour `y` au point $p$. Quelle est la partie du programme (la tranche, ou slice) qui a pu affecter la valeur de `y` ?

```
...
y:=1;
while (x>0) {
  y:=y*x;
  x:=x-1;
}
/* y ? */
```

```
...
y:=1;
while (x>0) {
  y:=y*x;
  x:=x-1;
}
/* y ? */
```

```
...
y:=1;
while (x>0) {
  y:=y*x;
  x:=x-1;
}
/* y ? */
```

# Le slicing, en Caml

```
let slicing graphe sommet varset =
    (* ajoute à slice les arcs accessibles à partir de sommet
       en arrière dans graphe qui ont pu modifier une variable dans varset *)
    if sommet ∈ slice
      then return;
    else let arcs_entrants = {p | p --instr--> sommet arc de graphe}
      in pour tout arc = (p --instr--> sommet) ∈ arcs_entrants :
        if varset ∩ Def(instr) ≠ ∅
          then slice := slice ∪ {arc};
                slicing graphe p ((varset \ Def(instr)) ∪ Use(instr))
        else slicing graphe p varset
slice := ∅; slicing graphe p {y};
```

où :

| $instr$ | $Def(instr)$ | $Use(instr)$ |
|---|---|---|
| $x := e$ | $\{x\}$ | $\mathrm{fv}(e)$ |
| $\texttt{input}\, x$ | $\{x\}$ | $\emptyset$ |
| $\texttt{skip}$ | $\emptyset$ | $\emptyset$ |
| $e$ vrai? | $\emptyset$ | $\mathrm{fv}(e)$ |
| $e$ faux? | $\emptyset$ | $\mathrm{fv}(e)$ |

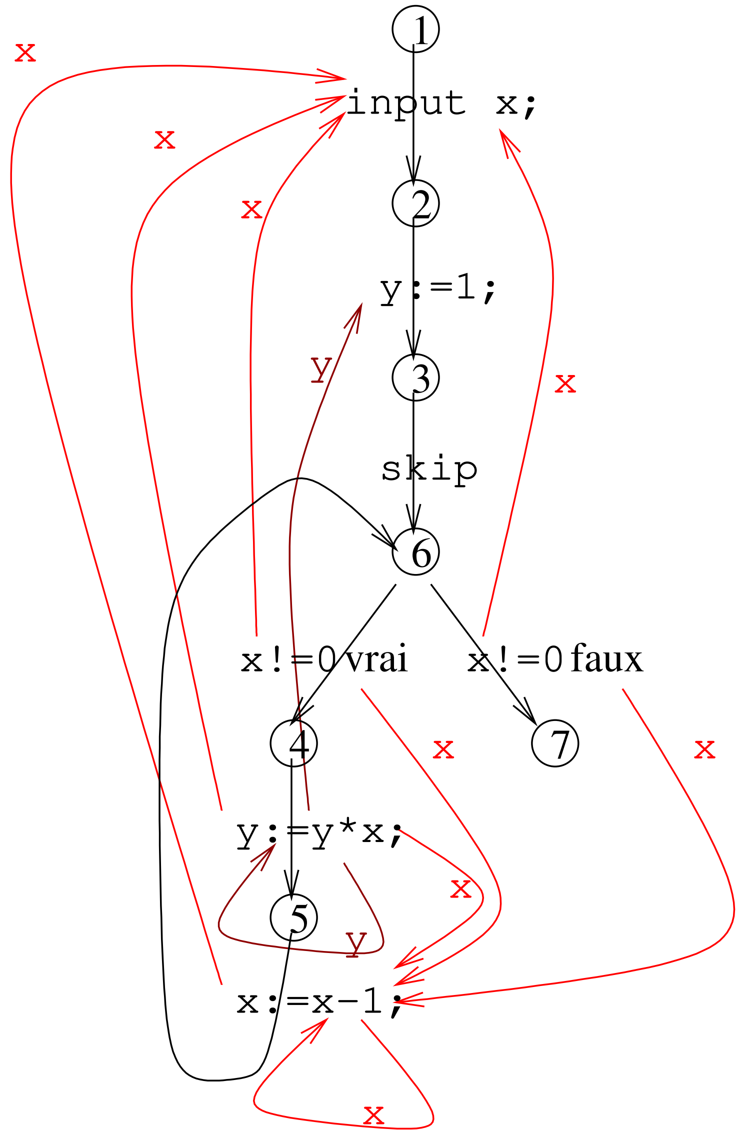
$\mathrm{fv}(e) \hat{=} \{$variables apparaissant dans $e\}$

## Graphes Use-Def

En fait, pour faire du slicing, on n'a pas besoin du graphe de contrôle, juste du graphe use-def, dont :

les nœuds sont les arcs du graphe de contrôle, et

les arcs $e_1 \xrightarrow{x} e_2$ sont ceux tels qu'il existe un chemin $\underbrace{\xrightarrow{instr_0}}_{e_2} p_1 \xrightarrow{instr_1} \ldots \xrightarrow{instr_n} p_n \underbrace{\xrightarrow{instr_{n+1}}}_{e_1}$ dans le graphe de contrôle avec $x \in Def(instr_0), x \notin Def(instr_1), \ldots, x \notin Def(instr_n)$, $x \in Use(instr_{n+1})$ :


1
input x;
2
y:=1;
3
skip
6
x!=0 vrai
x!=0 faux
4
7
y:=y*x;
5
x:=x-1;
x
y

## Améliorations du graphe use-def

Forme SSA (Static Single Assignment) : on s'arrange pour que toute instruction définisse au plus une variable ($\forall x \cdot \exists! p \overset{instr}{\longrightarrow} q \cdot x \in Def(instr)$).

– Nécessite une astuce pour les boucles (les nœuds $\phi$);

– Revient à compiler notre langage vers un langage purement fonctionnel;

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck, An efficient method of computing static single assignment form. *16th ACM Symposium on Principles of Programming Languages (POPL'89)*, 25–35, 1989.

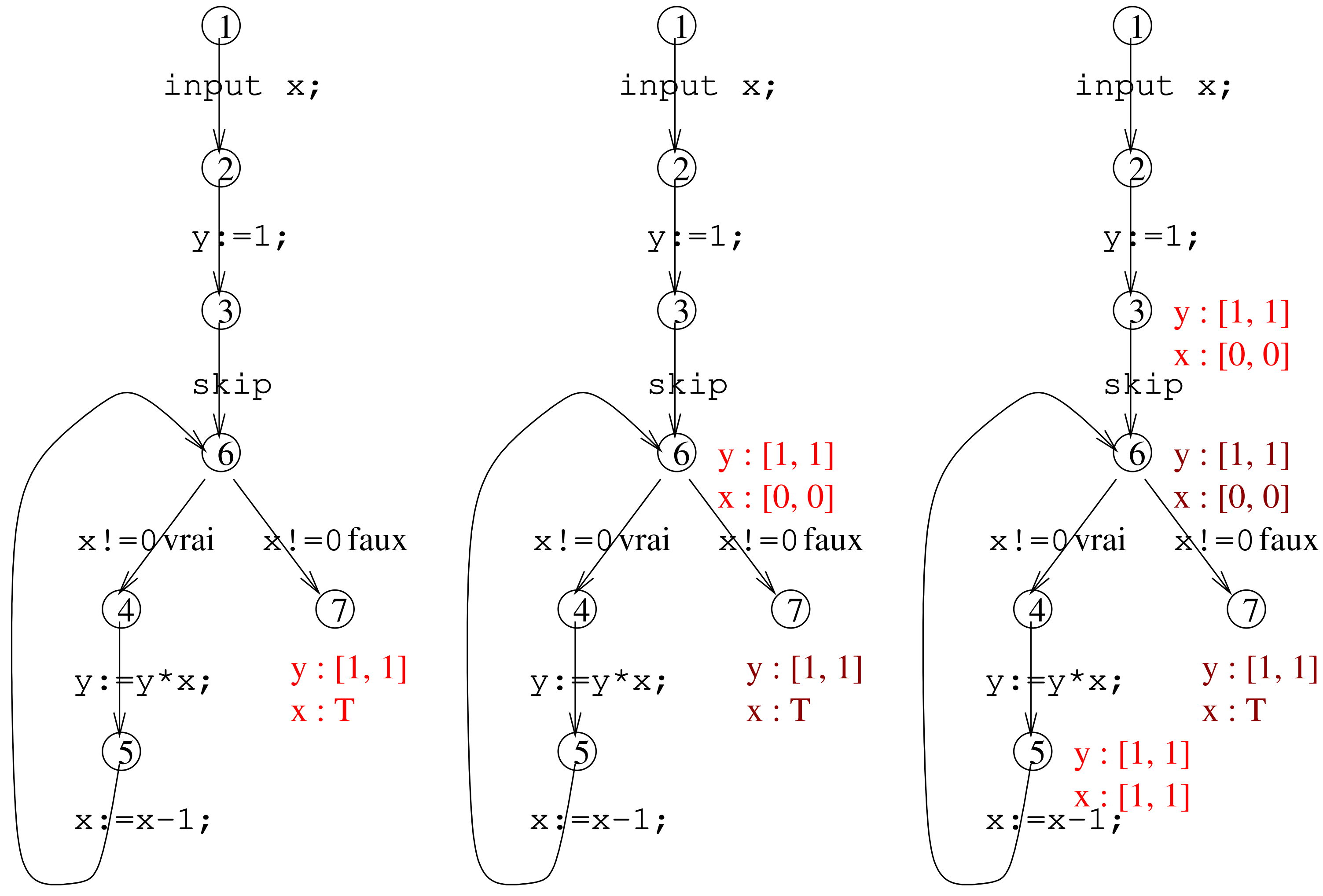
Value Dependence Graphs : code à la fois SSA et graphe de contrôle; améliore efficacité des analyses, mais production de code plus délicate.

Daniel Weise, Roger F. Crew, Michael Ernst, Bjarne Steensgaard, Value dependence graphs: representation without taxation, *21th ACM Symposium on Principles of Programming Languages (POPL'94)*, 297–310, 1994.

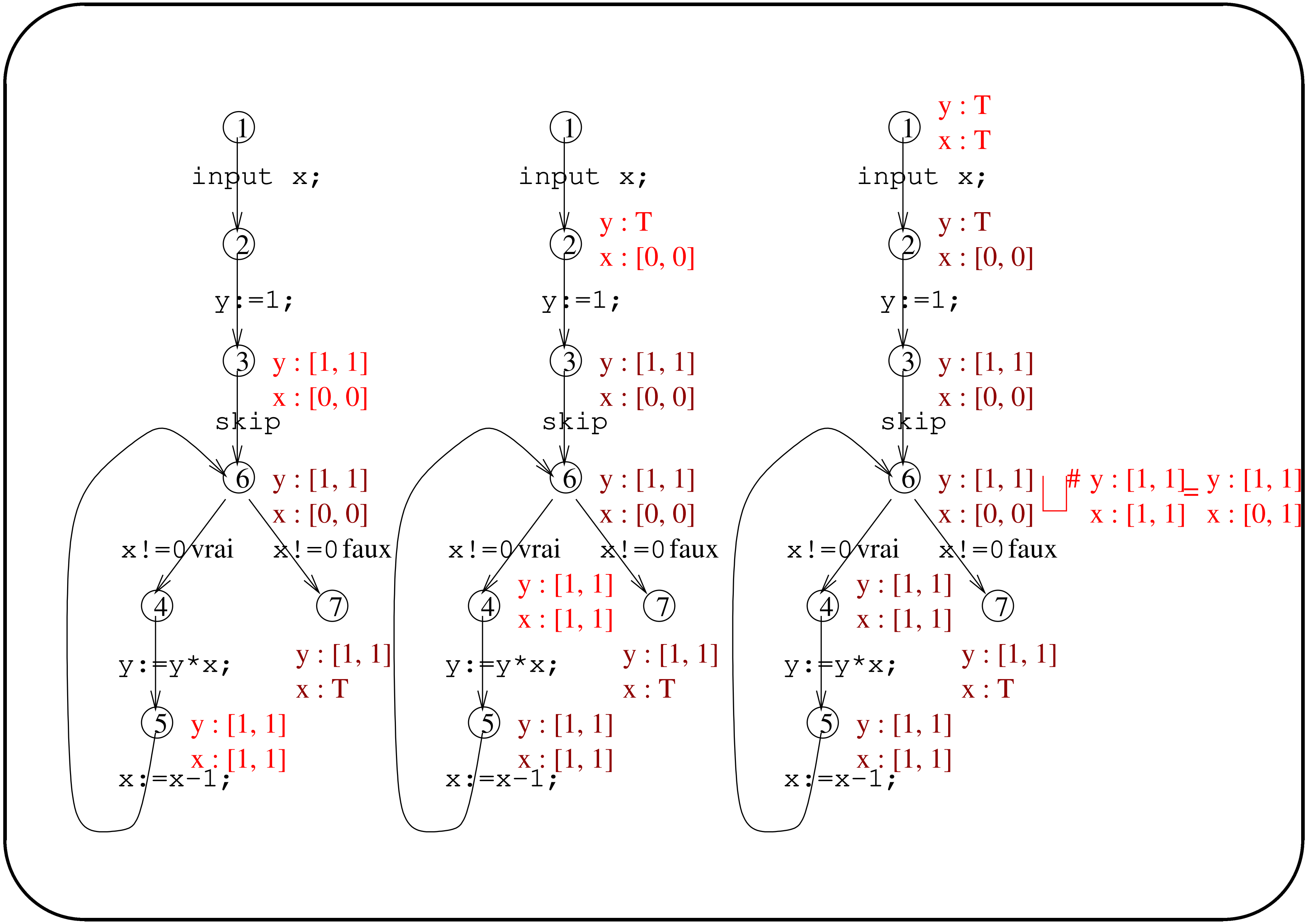
# Généralisation : le debugging abstrait (exemple)

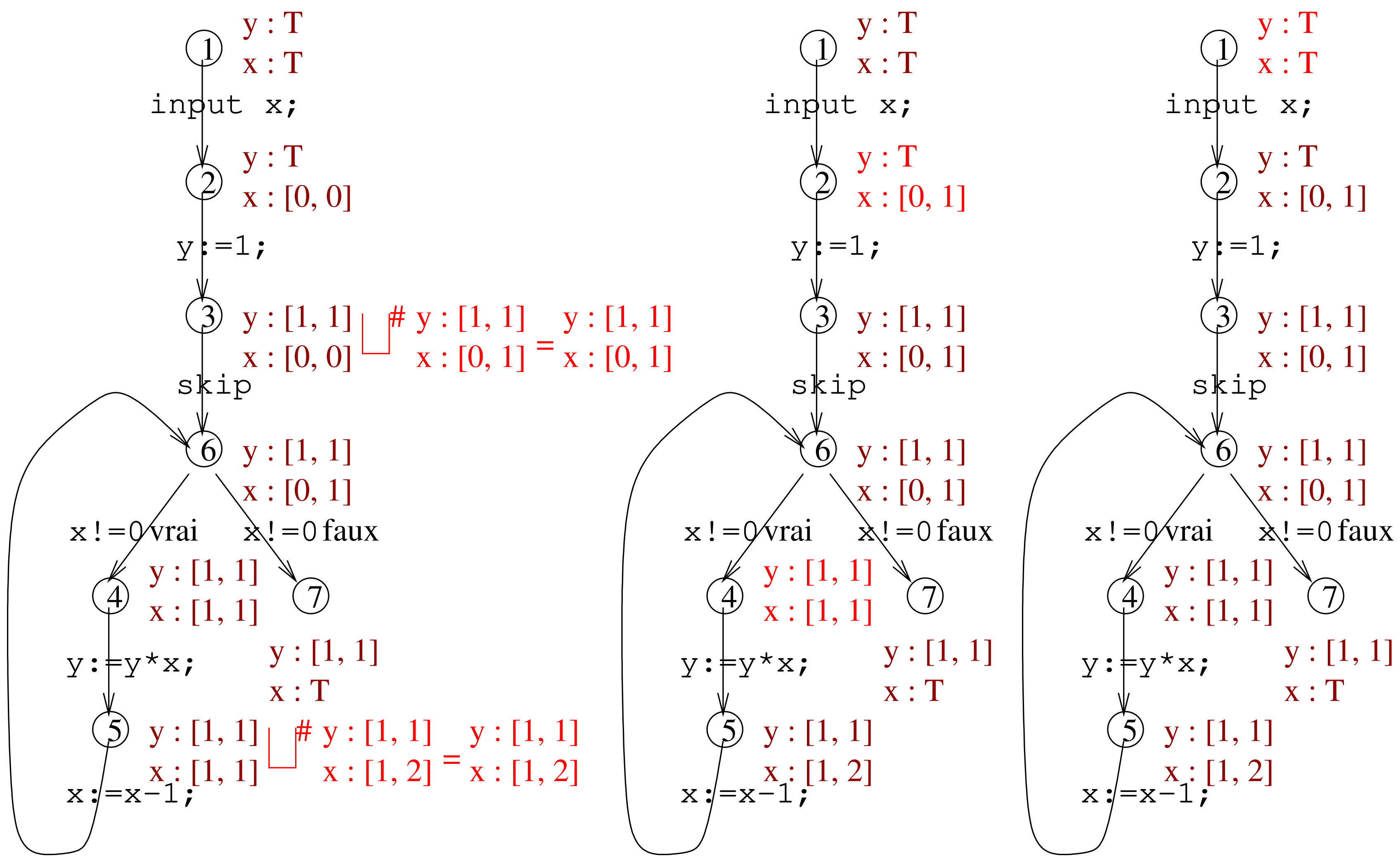
```
...
y:=1;
while (x!=0) {
  y:=y*x;
  x:=x-1;
}
```

Après avoir executé le programme, on trouve `y`$= 1$. Mais pour le programmeur `y` aurait dû valoir au moins 2. Quelles pouvaient être les valeurs de `x`, `y` avant la boucle qui aboutissent à ce résultat ?



1
input x;
2
y:=1;
3
skip
6
x!=0 vrai
x!=0 faux
4
7
y:=y*x;
y : [1, 1]
x : T
5
x:=x-1;
1
input x;
2
y:=1;
3
skip
6
y : [1, 1]
x : [0, 0]
x!=0 vrai
x!=0 faux
4
7
y:=y*x;
y : [1, 1]
x : T
5
x:=x-1;
1
input x;
2
y:=1;
3
y : [1, 1]
x : [0, 0]
skip
6
y : [1, 1]
x : [0, 0]
x!=0 vrai
x!=0 faux
4
7
y:=y*x;
y : [1, 1]
x : T
5
y : [1, 1]
x : [1, 1]
x:=x-1;


input x;
y:=1;
skip
x!=0 vrai
x!=0 faux
y:=y*x;
x:=x-1;
y : T
x : T
x : [0, 0]
y : [1, 1]
x : [1, 1]
# y : [1, 1] = y : [1, 1]
x : [1, 1] x : [0, 1]



y : T
x : T
input x;
y : T
x : [0, 0]
y:=1;
y : [1, 1]
x : [0, 0]
⊔# y : [1, 1]
x : [0, 1]
= y : [1, 1]
x : [0, 1]
skip
y : [1, 1]
x : [0, 1]
x!=0 vrai
x!=0 faux
y : [1, 1]
x : [1, 1]
y:=y*x;
y : [1, 1]
x : T
y : [1, 1]
x : [1, 1]
⊔# y : [1, 1]
x : [1, 2]
= y : [1, 1]
x : [1, 2]
x:=x-1;
y : T
x : T
input x;
y : T
x : [0, 1]
y:=1;
y : [1, 1]
x : [0, 1]
skip
y : [1, 1]
x : [0, 1]
x!=0 vrai
x!=0 faux
y : [1, 1]
x : [1, 1]
y:=y*x;
y : [1, 1]
x : T
y : [1, 1]
x : [1, 2]
x:=x-1;
y : T
x : T
input x;
y : T
x : [0, 1]
y:=1;
y : [1, 1]
x : [0, 1]
skip
y : [1, 1]
x : [0, 1]
x!=0 vrai
x!=0 faux
y : [1, 1]
x : [1, 1]
y:=y*x;
y : [1, 1]
x : T
y : [1, 1]
x : [1, 2]
x:=x-1;

## Remarques

Donc les seules valeurs de $\mathtt{x}$ qui pouvaient donner $\mathtt{y}= 1$ en sortie sont 0 et 1. C'est l'information exacte (ici, pas en général), et elle a été trouvée automatiquement en utilisant le treillis des intervalles.

En réalité, on aurait utilisé un élargissement sur l'arc arrière $5 \xrightarrow{\mathtt{x:=x-1}} 6$. Montrer que ça n'aurait pas changé le résultat, sauf qu'au point 2, on trouverait $x : [1, +\infty]$ au lieu de $x : [1, 2]$.

Note : sur l'arc $4 \xrightarrow{\mathtt{y:=y*x}} 5$, on a utilisé que si $\mathtt{x} \times \mathtt{y} = 1$ et $\mathtt{x} = 1$, alors $\mathtt{y} = 1$. C'est un arc inversible : on peut calculer la valeur de $\mathtt{y}$ avant en fonction de celle de $\mathtt{y}$ après l'instruction. À l'opposé, l'arc $2 \xrightarrow{\mathtt{y:=1}} 3$ est non inversible.

# La sémantique arrière

Pour faire du slicing ou du debugging abstrait, on a besoin de remonter l'exécution du programme à l'envers, en arrière.

C'est facile sur la sémantique concrète : les jugements sont de la forme $p_1, \rho_1 \vdash instr \Rightarrow p_2, \rho_2 \quad$ (si $cond[\![instr]\!](\rho_1, \rho_2)$). On peut donc, pour tout $p_2$ et tout $\rho_2$, trouver les prédécesseurs $p_1$ et les $\rho_1$ tels que $cond[\![instr]\!](\rho_1, \rho_2)$ soit vrai.

On obtient ainsi la sémantique collectante arrière $\mathcal{P}_{\leftarrow}(E)$ . . . tout marche comme aux transparents 71 et suivants en renversant la direction d'exécution des programmes !

# Cotraces

Une cotrace (d'exécution) $T$ arrivant sur $q \in \mathrm{PP}$, $R \in \mathbb{P}(\mathcal{E}nv)$ est une suite $(p_1^T, \rho_1^T), (p_2^T, \rho_2^T), \ldots, (p_N^T, \rho_N^T)$ telle que :

$$p_1^T, \rho_1^T \vdash instr_1 \Rrightarrow p_2^T, \rho_2^T \vdash instr_2 \Rrightarrow \ldots \Rrightarrow p_N^T, \rho_N^T$$

et $p_N^T = q$, $\rho_N^T \in R$.

L'ensemble des environnements $Env_{q,R}(p)$ au point de programme $p$ qui peuvent mener à $q$ avec un environnement $\rho \in R$ est $\{\rho | \exists T$ cotrace arrivant sur $q, R \cdot \rho = \rho_1^T \wedge p_1^T = p\}$.

Le but de l'interprétation abstraite arrière sera de calculer $Env^\sharp \in \mathcal{E}nv^\sharp$ tel que pour tout point de programme $p$, $Env_{q,R}(p) \subseteq \gamma_{NR}(Env^\sharp(p))$ (correction), c'est-à-dire $\forall \rho \in Env_{q,R}(p) \cdot \forall x \in \mathcal{V}ar \cdot \rho(x) \in \gamma(Env^\sharp(p)(x))$.

## Sémantique collectante arrière

Supposons que l'on ait une estimation $E$ de $Env_{q,R} \hat{=} \lambda p \cdot Env_{q,R}(p)$.

On peut calculer une meilleure estimation $E'$ de $Env_{q,R}$ en voyant ce qui se passe avant une étape de calcul :

$$E'(p) \hat{=} E(p) \cup \{\rho \mid \exists p_2, \rho_2 \in E(p_2), instr \cdot p, \rho \vdash instr \Rightarrow p_2, \rho_2\}$$

$$[\cup R \qquad (\text{si } p = q)]$$

($E(p)$ plus résultats d'une étape de calcul)

[plus $R$ si $p = q$ : condition d'arrivée]

$E'$ est une fonction de $E$, la sémantique collectante arrière $\mathcal{P}_{\leftarrow,q,R}(E)$ du programme.

## Une caractérisation de $Env_{q,R}$

**Théorème :** $Env_{q,R} \hat{=} \lambda p \cdot Env_{q,R}(p) = lfp_{\subseteq^*}(\mathcal{P}_{\leftarrow,q,R})$, où $lfp_{\subseteq^*}$ est l'opérateur "plus petit point fixe" et $E \subseteq^* E'$ ssi pour tout $p$, $E(p) \subseteq E'(p)$.
**Démonstration :** comme au transparent 78.

Sémantique abstraite arrière : $Env^\sharp_{q,R} \hat{=} lfp_{\sqsubseteq^{\sharp *}}(\mathcal{P}_{\leftarrow,q,R}{}^\sharp)$, où

$$\mathcal{P}_{\leftarrow,q,R}{}^\sharp(E^\sharp)(p) \hat{=} E^\sharp(p) \sqcup^\sharp \bigsqcup{}^\sharp \{cond^\sharp[\![instr]\!]_\leftarrow \rho_2^\sharp \mid p \xrightarrow{instr} p_2, \rho_2^\sharp = E^\sharp(p_2)\}$$
$$\sqcup^\sharp \lambda p \cdot \begin{cases} \alpha_{NR}(R) & \text{si } p = q \\ \bot & \text{sinon} \end{cases}, \text{ et :}$$

$$cond^\sharp[\![instr]\!]_\leftarrow \rho^\sharp \quad \hat{=} \quad ? \quad \ldots \text{ cf. transparent 161}$$

# L'analyseur arrière (cf. transparent 141)

$\mathtt{let\ rec\ cfg\_type\_arr}\ E^\sharp\ \rho^\sharp\ \mathtt{graphe}\ \mathtt{sommet}\ \mathtt{elargir} =$

$\quad \mathtt{if}\ E^\sharp(\mathtt{sommet}) = \rho^\sharp$

$\qquad \mathtt{then\ return;}$

$$\mathtt{else}\ E^\sharp(\mathtt{sommet}) := \begin{cases} E^\sharp(\mathtt{sommet})\nabla^*\rho^\sharp & \text{si } \mathtt{elargir} = \mathtt{true} \\ E^\sharp(\mathtt{sommet}) \sqcup^{\sharp *} \rho^\sharp & \text{sinon} \end{cases} ;$$

$\qquad \mathtt{let}\ \mathtt{arcs\_entrants} = \{p \mid p \overset{instr}{\longrightarrow} \mathtt{sommet} \text{ arc de } \mathtt{graphe}\}$

$\qquad \mathtt{in}$ pour tout $(p \overset{instr}{\longrightarrow} \mathtt{sommet}) \in \mathtt{arcs\_entrants}$ :

$\qquad\quad \mathtt{cfg\_type\_arr}\ E^\sharp\ (cond^\sharp[\![instr]\!]_{\leftarrow}\rho^\sharp)\ \mathtt{graphe}\ p\ (p \geq \mathtt{sommet});;$

$\mathtt{cfg\_type\_arr}\ (\lambda p, x \cdot \bot)\ R^\sharp\ \mathtt{graphe}\ q\ \mathtt{false};;$ (* $R^\sharp = \alpha(R)$ *)

# Calcul de $cond^\sharp[\![instr]\!]_{\leftarrow}\rho^\sharp$, cas faciles

$$
\begin{array}{rcl}
cond^\sharp[\![\texttt{input}\, x]\!]_{\leftarrow}\rho^\sharp & \hat{=} & \rho^\sharp[x \leftarrow \top] \\
cond^\sharp[\![\texttt{skip}]\!]_{\leftarrow}\rho^\sharp & \hat{=} & \rho^\sharp
\end{array}
$$

Le cas des affectations $x{:=}e$ et celui des tests $e$ vrai?/$e$ faux? est plus compliqué, et n'a pas de solution systématique générale comme en analyse avant.

## Calcul de $cond^\sharp[\![x{:=}e]\!]_{\leftarrow}\rho^\sharp$, connaissant $v^\sharp \hat{=} \rho^\sharp(x)$

$= \rho^\sharp[x \leftarrow v_0^\sharp]$, où $v_0^\sharp$ est une valeur abstraite (aussi précise que possible) telle que $\{v_0 \mid \mathcal{E}[\![e]\!](\rho[x := v_0]) \in \gamma(v^\sharp), \rho \in \gamma_{NR}(\rho^\sharp)\} \subseteq \gamma(v_0^\sharp)$ (correction).

- Solution idéale : poser

$$v_0^\sharp \hat{=} \alpha\{v_0 \mid \mathcal{E}[\![e]\!](\rho[x := v_0]) \in \gamma(v^\sharp), \rho \in \gamma_{NR}(\rho^\sharp)\}$$

… mais c'est incalculable en général.

- Solution la plus courante : reconnaître la forme syntaxique de $e$.

  Exemple : cas de contraintes arithmétiques, treillis des intervalles :

  – si $e$ est de la forme $ax + b$ (à simplification près), $x$ n'apparaissant pas dans $a$ ou $b$, alors $v_0^\sharp$ = valeur abstraite de $(v - b)/a$ $= (v^\sharp -^\sharp \mathcal{E}[\![b]\!]^\sharp\rho^\sharp)/^\sharp\mathcal{E}[\![a]\!]^\sharp\rho^\sharp$; (cas inversible)

  – sinon, $v_0^\sharp \hat{=} \top$ (cas non linéaires + cas non inversibles).

# Itérations décroissantes locales

Solution plus précise, par itération de point fixe.

Philippe Granger, Improving the results of static analyses of programs by local decreasing iterations, *12th Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'92)*, Springer Verlag Lecture Notes in Computer Science 652, 68–79, 1992

Exemple : `x:=2*x*x-1`, $v^\sharp = ([0,2])$ :
Init : $v^\sharp_{0,0} := ([-\infty,+\infty])$ (pas $\top$; ex : pourquoi le peut-on ?);
Étape 1 : on résout $2 * v^\sharp_{0,0} * x - 1 = ([0,2])$ en $x \Rightarrow x = v^\sharp_{0,1} \hat{=} ([-1,1])$;
Étape 2 : on résout $2 * v^\sharp_{0,1} * x - 1 = ([0,2])$ en $x \Rightarrow x = v^\sharp_{0,2} \hat{=} ([-1,1])$.
Donc $v^\sharp_0 = ([-1,1])$.

En général, on prend pour $v^\sharp_0$ la limite d'une suite décroissante de $v^\sharp_{0,i}$.
Peut nécessiter de forcer la convergence à l'aide d'un élargissement dual $\overline{\nabla}$ (= élargissement pour l'ordre inverse).

## Calcul de $cond^\sharp[\![e \textbf{ vrai?}]\!]_{\leftarrow}\rho^\sharp$

$= \rho_0^\sharp$ (le plus précis possible) tel que
$\{\rho_0 \mid \rho_0 \in \gamma_{NR}(\rho^\sharp) \text{ et } \mathcal{E}[\![e]\!]\rho_0 = \texttt{true})\} \subseteq \gamma_{NR}(\rho_0^\sharp)$ (correction).

- Solution idéale : $\rho_0^\sharp \hat{=} \alpha\{\rho_0 \mid \rho_0 \in \gamma_{NR}(\rho^\sharp) \text{ et } \mathcal{E}[\![e]\!]\rho_0 = \texttt{true})\}$

  … mais c'est incalculable en général.
- Solution la plus courante : par reconnaissance syntaxique de $e$.

  Exemple : cas de contraintes arithmétiques, treillis des intervalles :

  – si $e$ est de la forme $ax + b = 0$, $a \neq 0$, $x$ n'apparaissant pas dans $a$ ou $b$, alors $\rho_0^\sharp \hat{=} \rho^\sharp[x \leftarrow \rho^\sharp(x) \sqcap^\sharp \mathcal{E}[\![-b \;/\; a]\!]\rho^\sharp]$ (pour une certaine variable $x$ $(*)$);

  – si $e = (b = 0)$, $0 \notin \gamma(\mathcal{E}[\![b]\!]^\sharp\rho^\sharp)$, alors $\rho_0^\sharp = \bot$ (contradiction);

  – si $e = (b = 0)$, sinon, $\rho_0^\sharp \hat{=} \rho^\sharp$.
- Ou par itérations décroissantes locales (utiliser $(*)$).

## Remarques

Le slicing, c'est l'analyse arrière basée sur le treillis des initialisations (exercice).

On peut aussi améliorer les analyses avant des tests en faisant une analyse arrière juste après pour raffiner les cas de succès du test.

**Ex :** $6 \xrightarrow{x=0 \text{ vrai?}} 4$. Supposons qu'en 6, `x` : $(\![-\infty, +\infty]\!)$.
Analyse avant $\Rightarrow$ en 4, `x` : $(\![-\infty, +\infty]\!)$.
Analyse arrière $\Rightarrow$ si on a pu arriver en 4, alors en 6, `x` : $(\![0, 0]\!)$.
Analyse avant $\Rightarrow$ donc en 4, `x` : $(\![0, 0]\!)$.

En général, on peut alterner analyses avant et arrière pour obtenir des analyses plus précises que chacune d'elle séparément.