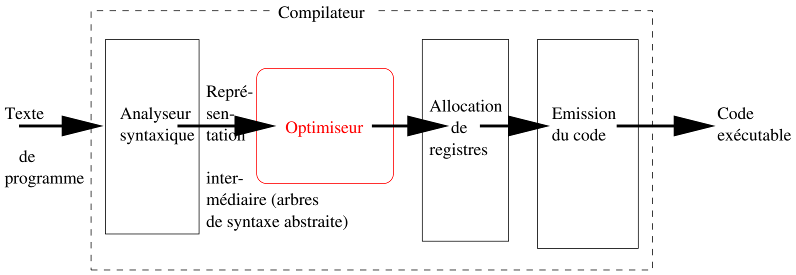
Projet d'ouvrage sur la programmation

Thérèse Hardin, Véronique Viguié

version v1.0e, 24 juin 2000

1. Analyse statique

L'analyse statique de programme désigne toute technique permettant de déduire des propriétés sur tout programme dans un langage de programmation donné, et ce par algorithme. Un exemple typique et courant est celui du *compilateur optimisant*, par exemple n'importe quel compilateur du langage C auquel on aura fourni l'option `-O`, le compilateur `ocamlopt` pour Objective Caml, etc. L'architecture typique d'un tel compilateur est :



Lorsqu'on lance le compilateur sur le fichier texte (dit *source*) du programme à compiler, le compilateur commence par effectuer l'analyse lexicale et statique du source (voir chapitre ??), et construit un arbre de syntaxe abstraite. En principe, le compilateur pourrait retraduire cet arbre directement sous forme de code machine. Par exemple, le morceau de programme C suivant :

```
int f (int m, int n) { int p = m+n;  return p+3; }
```

se compile sous forme du code suivant, si on utilise `gcc` version 2.8.1, sans option, sur une machine à processeur Intel Pentium sur Linux :

	Mnémoniques assembleur	Explication	Octets émis
1	`movl 8(%ebp),%eax`	mettre `m` dans `%eax`	139, 69, 8
2	`movl 12(%ebp),%edx`	mettre `n` dans `%edx`	139, 85, 12
3	`leal (%edx,%eax),%ecx`	calculer `%ecx`:=`%eax`+`%edx`	141, 12, 2
4	`movl %ecx,-4(%ebp)`	ranger `%ecx` dans `p`	137, 77, 252
5	`movl -4(%ebp),%edx`	mettre `p` dans `%edx`	139, 85, 252
6	`addl $3,%edx`	ajouter 3 à `%edx`	131, 194, 3
7	`movl %edx,%eax`	retourner `%edx`	137, 208

Les notations `%ebp`, `%eax`, `%ecx`, `%edx` désignent des registres du processeur, qui sont les endroits où le processeur fait ses calculs. On voit que chaque instruction du programme `f` correspond à un groupe de lignes dans le code : `p = m+n` forme les quatre premières lignes, l'addition avec 3 les deux suivantes, et le `return` (qui consiste à mettre le résultat dans le registre de retour `%eax`) la dernière ligne. (Nous n'insisterons pas plus sur la signification précise des mnémoniques.) Un compilateur simple peut donc se contenter d'écrire dans un fichier de sortie (le code *exécutable*) les nombres 139, 69, 8, 139, 85, 12, 141, 12, 2, 137, 77, 252. Cette suite de nombres sera interprétée par le processeur comme une série d'ordres: mettre `m` dans `%eax`, mettre `n` dans `%edx`, etc.

Mais un tel code exécutable n'est pas le plus efficace possible. En particulier, la quatrième ligne range `%ecx` dans `p`, pour le relire immédiatement dans `%edx`. On pourrait donc remplacer la ligne 5 par une instruction qui transférerait directement le contenu de `%ecx` dans `%edx`, ce qui serait plus rapide, les opérations dans les registres étant beaucoup plus rapides qu'entre mémoire (`p`) et registres. En fait, on pourrait même éliminer la ligne 4 entièrement, parce que le programme n'a plus jamais besoin de réutiliser le contenu de la case mémoire `p`. Ce genre de transformations est le rôle de l'*optimiseur*.

Mais l'optimiseur ne peut pas se contenter d'éliminer la ligne 4 aveuglément. Il doit d'abord s'assurer que `p` ne sera plus jamais utilisé dans le reste du programme. Sinon, la variable `p` est *vivante*, et la ligne 4 doit être maintenue. Cette propriété du programme ne peut pas être décidée localement, c'est-à-dire en regardant juste les instructions au voisinage de la ligne 4. L'optimiseur doit explorer tout le reste du programme pour découvrir si `p` est vivante ou non. C'est ici que les techniques d'analyse statique sont utilisées.

De plus en plus, les techniques d'analyse statique sont utilisées non pour assurer la correction de transformations de programmes, comme ci-dessus, mais pour détecter des erreurs de programmation. Dans certains cas, l'optimiseur du compilateur suffit : par exemple, si on lance `gcc -O -Wall` sur le programme suivant, on obtiendra le message `In function 'g': warning: 'j' might be used uninitialized in this function`, signalant que j n'a pas été initialisée à sa première utilisation, ce qui produira un résultat aléatoire.

```
int g (int i) { int j, k;   k = i+j;  return k; }
```

Plus généralement, l'analyse statique permet de détecter des erreurs à l'exécution, par exemple des débordements de tableaux (accéder à l'élément 25 d'un tableau dont le dernier élément est le numéro 24) ou des erreurs de type (vouloir exécuter le code d'une fonction f, alors que f n'est pas une fonction, mais un entier). Ces erreurs sont à l'origine de bogues graves, incluant la corruption de données, les trous de sécurité, l'envoi de commandes non voulues à des dispositifs externes ou des cas de non-déterminisme incontrôlé (ces deux derniers cas étant les causes principales de l'explosion du premier vol Ariane 5 [LMR+98]). Quelques compagnies proposent aujourd'hui des outils d'analyse de la sûreté des logiciels par analyse statique, notamment [Pol00, Abs00].

1.1. *Quelques résultats théoriques*

L'analyse de programme, il est important de le noter, doit être effectuée en fonction de valeurs *inconnues* des données initiales. Par exemple, dans la fonction `g` ci-dessus, on veut pouvoir dire que `j` ne sera jamais initialisée, quelle que soit la valeur du paramètre `i`. Il est donc irréaliste de chercher à vérifier les propriétés d'un programme (ici, que `j` soit non initialisée à son premier usage) en testant tous les cas possibles (toutes les valeurs possibles de `i`). Nul doute que le test est utile, mais pour avoir des réponses sûres, il ne suffit pas : il nous faudra inventer des algorithmes qui découvrent ces propriétés sur des programmes sans les exécuter. C'est pourquoi on parle d'analyse *statique* par opposition au caractère dynamique de l'exécution d'un programme.

Malheureusement, il n'existe aucun algorithme permettant de décider une propriété quelconque digne d'intérêt. C'est le théorème de Rice, dont l'énoncé exact est : toute propriété extensionnelle non triviale de programmes est indécidable — *extensionnelle* signifiant que la propriété doit ne dépendre que de la *sémantique* du programme, et pas de son nombre de lignes ou du nombre d'imbrications de boucles utilisées, par exemple, et *non triviale* signifiant ni vraie de tous les programmes ni fausse de tous les programmes.

Qu'il n'existe pas d'algorithme décidant une propriété P signifie que toute procédure qui décide P doit boucler indéfiniment sur au moins certains programmes, ou bien que tout algorithme (procédure qui termine) d'analyse doit se tromper sur au moins certains programmes. Heureusement, on peut choisir dans quel sens l'algorithme d'analyse statique va se tromper. Dans l'exemple des variables non initialisées, on va choisir typiquement un optimiseur qui est toujours correct quand il découvre qu'une variable est initialisée, mais qui peut se tromper quand il découvre une variable qu'il croit non initialisée. Par exemple, `gcc -O -Wall` version 2.8.1 se trompe sur le programme suivant :

```
int h2 (int i) { int j = 0;  int k;
                 if (i*(j-1)+i==0) k = 2;
                 return k; }
```

et annonce que `k` est utilisée (par `return`) à un point où elle n'a peut-être pas encore été initialisée : `warning: 'k' might be used uninitialized in this function`. L'optimiseur n'a pas vu que comme `j` vaut 0, le test `if` est vrai, ce qui force l'initialisation de `k`.

Comme on le voit, que l'optimiseur se trompe n'est pas grave : il vaut mieux être trop prudent que pas assez. Cette philosophie est la base de l'*interprétation abstraite*, qui forme le fondement théorique des techniques d'analyse statique.

1.2. *Une approche intuitive de l'interprétation abstraite*

En première approche, l'interprétation abstraite est un mode de calcul non standard, où les vraies valeurs (les valeurs *concrètes*) sont remplacées par des symboles (les valeurs *abstraites*). À cause des problèmes d'indécidabilité, on devra parfois forcer une perte d'information, faire des *approximations*.

Considérons par exemple l'expression de programme `x*x+y*y`, où `x` et `y` sont des nombres réels à virgule flottante. Quelles que soient les valeurs de `x` et `y`, `x*x+y*y` est positif ou nul. Les règles de calcul standard (concrètes) nous permettent de le vérifier sur tous les cas; par exemple, si $\mathtt{x} = 3$ et $\mathtt{x} = -4$, alors `x*x+y*y` vaut 25, qui est effectivement positif ou nul. Mais on peut arriver à la conclusion que le résultat est *toujours* positif ou nul sans énumérer tous les cas, par la règle des signes. On remplace les valeurs concrètes par les deux signes $\geq \mathbf{0}$ et $\leq \mathbf{0}$, et on utilise les tables :

+ abstrait	$\geq \mathbf{0}$	$\leq \mathbf{0}$
$\geq \mathbf{0}$	$\geq \mathbf{0}$	$\top$
$\leq \mathbf{0}$	$\top$	$\leq \mathbf{0}$

$*$ abstrait	$\geq \mathbf{0}$	$\leq \mathbf{0}$
$\geq \mathbf{0}$	$\geq \mathbf{0}$	$\leq \mathbf{0}$
$\leq \mathbf{0}$	$\leq \mathbf{0}$	$\geq \mathbf{0}$

où $\geq \mathbf{0}$ signifie "positif ou nul", $\leq \mathbf{0}$ signifie "négatif ou nul", et où $\top$ signifie "on ne sait pas". Le signe du résultat de l'addition d'un nombre positif et d'un nombre négatif ne peut être obtenu en examinant simplement le signe de ces deux nombres. Dans ce cas la valeur renvoyée est $\top$, et c'est ici que ce calcul abstrait simple perd de l'information. Observons notamment que ce dernier ne peut pas déduire que si `x` est $\geq \mathbf{0}$ et `y` est $\leq \mathbf{0}$, alors `x*x+2*x*y+y*y` est $\geq \mathbf{0}$, ni que `x*x` est $\geq \mathbf{0}$ lorsque `x` est $\top$. Rappelons qu'une telle perte d'information est nécessaire si l'on veut que le calcul abstrait retourne un résultat.

Ce qui est plus important que la perte ou la non-perte d'information, c'est que le calcul abstrait est *correct* par rapport au calcul concret : pour toute expression e, si e est $\geq \mathbf{0}$ en abstrait, alors $e \geq 0$ en concret, et si e est $\leq \mathbf{0}$ en abstrait, alors $e \leq 0$ en concret. L'interprétation abstraite est un cadre qui permet de produire des calculs abstraits corrects de façon presque systématique. La question de la précision du calcul abstrait (la rareté des cas de perte d'information) est autre, et demande de l'expérience.

Nous allons illustrer les techniques d'interprétation abstraite sur un langage impératif jouet. On part d'une *syntaxe* :

$$
\begin{array}{rcll}
x, y, z, \ldots \in \mathcal{V}ar & \hat{=} & \{\ldots, \mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots\} & \text{variables} \\
a, m, n, \ldots \in \mathcal{C}st & \hat{=} & \mathbb{B} \mid \mathbb{Z} \mid \mathcal{S}tring & \text{constantes} \\
\mathbb{B} & \hat{=} & \{\mathtt{true}, \mathtt{false}\} & \text{booléens} \\
\mathbb{Z} & \hat{=} & \{\ldots, -2, -1, 0, 1, 2, \ldots\} & \text{entiers} \\
\mathcal{S}tring & \hat{=} & \{\text{`` ''}, \text{``abc''}, \ldots\} & \text{chaînes}
\end{array}
$$

$$
\begin{array}{rcll}
e, \ldots \in \mathcal{E}xpr & \hat{=} & x \mid a & \text{expressions} \\
& \mid & e \,\mathtt{+}\, e \mid \mathtt{-}e \mid e * e \mid e \,/\, e \mid e \,\mathtt{=}\, e \mid \mathtt{not}\, e & \\
& & & \text{commandes} \\
c, \ldots \in \mathcal{C}om & \hat{=} & x := e & \text{affectation} \\
& \mid & \mathtt{input}\, x & \text{lecture de valeur au clavier} \\
& \mid & \mathtt{if}\, e\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2 & \text{conditionnelle} \\
& \mid & \mathtt{while}\, e\ \{c\} & \text{boucle} \\
& \mid & c_1 ; c_2 & \text{séquence} \\
& \mid & \mathtt{skip} & \text{absence d'action}
\end{array}
$$

Cette syntaxe se décrit bien à l'aide des déclarations Caml suivantes :

```
type var = VAR of string;;
type cst = B of bool | Z of int | R of float | S of string;;
type expr = V of var | C of cst
   | PLUS of expr * expr | MOINS of expr      | FOIS of expr * expr
   | DIV of expr * expr  | EGAL of expr * expr | NON of expr;;
type com = SET of var * expr | INPUT of var    | IF of expr * com * com
   | WHILE of expr * com     | SEQ of com * com | SKIP;;
```

On est alors tenté d'écrire un interprète abstrait calculant de façon approchée les signes des variables d'un programme (de type `com`) comme suit :

```
type signe = POS | NEG | TOP;; (* Les signes ≥ 0, ≤ 0, ⊤. *)
type signenv = var -> signe;; (* Environnements abstraits
                                 = fonctions variable ↦ signe. *)
(* On calcule maintenant le signe de e dans l'environnement rho par : *)
let rec expr_sign (rho:signenv) (e:expr) : signe = match e with
       V x -> rho x (* Signe de x donné par l'environnement ρ. *)
     | C c -> (match c with
                 Z n -> if n>=0 then POS else NEG                (1)
               | _ -> TOP)                                       (2)
     | PLUS (e1, e2) -> (match (expr_sign rho e1, expr_sign rho e2) with
       (POS, POS) -> POS | (NEG, NEG) -> NEG | _ -> TOP)
     | FOIS (e1, e2) -> (match (expr_sign rho e1, expr_sign rho e2) with
       (TOP, _) -> TOP | (_, TOP) -> TOP | (POS, POS) -> POS | _ -> NEG)
     | MOINS e1 -> (match expr_sign rho e1 with
                      POS -> NEG | NEG -> POS | TOP -> TOP)
     | DIV (e1, e2) -> (match (expr_sign rho e1, expr_sign rho e2) with
                          (POS, POS) -> POS                      (3)
                        | _ -> ...)
     | EGAL (e1, e2) -> TOP | NON e -> TOP;;
(* On calcule l'environnement des signes com_sign rho c qui sera effectif après
   l'exécution du programme c lorsqu'on part de l'environnement de signes rho: *)
let rec com_sign (rho:signenv) (c:com) : signenv = match c with
       SET (x, e) -> update rho x (expr_sign rho e)
     | INPUT x -> update rho x TOP
     | IF (e, c1, c2) -> sup (com_sign rho c1) (com_sign rho c2)       (4)
     | WHILE (e, c) -> sup rho (com_sign rho (SEQ (c, WHILE (e, c)))) (5)
     | SEQ (c1, c2) -> com_sign (com_sign rho c1) c2
     | SKIP -> rho;;                                                   (6)
```

où `update rho x v` retourne l'environnement qui est le même que `rho`, sauf que c'est le signe `v` qui est associé à `x`.

Ce premier prototype d'analyseur statique souffre de certains défauts :

- Ligne (1) : prenons le cas où `x` vaut 0, alors `expr_sign` calcule son signe comme étant $\geq \mathbf{0}$, alors que $\leq \mathbf{0}$ serait aussi valide. Une conséquence est qu'`expr_sign` va retourner $\top$ et non $\leq \mathbf{0}$ pour le signe de $0 + (-1)$ (`PLUS (Z 0, Z -1)`). C'est une perte d'information *absurde*.

- Ligne (2) : prendre le signe d'une chaîne ou d'un booléen n'a aucun sens; dans ces cas, `expr_sign` retourne `TOP`, c'est-à-dire $\top$ ("on ne sait pas"), mais il semble qu'une nouvelle valeur de signe, notons-la $\bot$ ("on sait très bien, au contraire, qu'on est dans une situation absurde") soit nécessaire. Retourner $\top$ à la place de $\bot$ est-il correct ?
- Ligne (3) : l'expression $1\ /\ 0$ (`DIV (Z 1, Z 0)`) a un signe évalué à $\geq \mathbf{0}$, mais le signe de $1\ /\ 0$, c'est-à-dire de l'erreur "division par zéro", n'est ni $\geq \mathbf{0}$, ni $\leq \mathbf{0}$, c'est l'absurdité $\bot$. Utiliser $\geq \mathbf{0}$ à la place de $\bot$ est-il correct ? Si oui, pourquoi utiliser $\geq \mathbf{0}$ ici, mais $\top$ dans le cas (2) ?
- Ligne (4) : on ne peut pas savoir quelle branche du test, `c1` ou `c2`, va être executée : on est obligé de calculer les signes sur les deux branches, et de combiner les résultats sur les deux branches, à l'aide d'une fonction `sup` à définir. Quelle doit être sa définition ?
- Ligne (5) : le cas du `while` a été écrit en remarquant que $\texttt{while}\, e\{c\}$ a la même sémantique que $\texttt{if}\ e\ \texttt{then}\,(c; \texttt{while}\, e\{c\})\ \texttt{else}\,\texttt{skip}$. Mais de même que l'analyseur `com_sign` ne sait pas tester quelle branche d'un `if` est prise, il ne sait pas si la boucle termine ou pas. En particulier l'analyseur tel qu'il est *boucle indéfiniment* dès qu'il tombe sur une boucle `while` : quel genre de test d'arrêt peut-on imposer pour résoudre ce problème ?
- Ligne (6) : l'analyseur ne donne des informations que sur les environnements à la fin du programme (s'il termine), pas sur les points de programmes intermédiaires. Mais on aimerait pouvoir vérifier des assertions de la forme "à la ligne n, la valeur de x est $\geq \mathbf{0}$".

Pour les points (2) et (3) notamment, on aura besoin de *sémantiques*, et d'une théorie permettant de relier des sémantiques (ici, la sémantique concrète qui dit comment le programme s'exécute réellement, et la sémantique abstraite des signes). C'est elle que nous présentons maintenant.

1.3. *Correspondance abstrait/concret, correspondances de Galois*

Commençons par décrire la sémantique formelle *concrète* des expressions en style dénotationnel $\mathcal{E}[\![e]\!]\rho$ des expressions e dans un environnement ρ (ci-dessous). Notons que le résultat de $\mathcal{E}[\![e]\!]\rho$ peut être ou une valeur de $\mathcal{V}al$, ou une erreur, et que nous avons choisi de séparer les erreurs en deux sortes, les erreurs de *typage* $\boxed{\mathrm{W}}$ (par exemple le résultat de prendre la négation d'une chaîne) et les erreurs à l'exécution $\boxed{\mathrm{E}}$ (par exemple une division par zéro).

$$
\begin{array}{rcll}
v, \ldots \in \mathcal{V}al & \hat{=} & \mathbb{B} \mid \mathbb{Z} \mid \mathcal{S}tring & \text{valeurs} \\
\rho, \ldots \in \mathcal{E}nv & \hat{=} & \mathcal{V}ar \to \mathcal{V}al \cup \{\boxed{\mathrm{E}}, \boxed{\mathrm{W}}\} & \text{environnements}
\end{array}
$$

$$
\begin{array}{rcll}
\mathcal{E}[\![\_]\!]\_ & : & \mathcal{E}xpr \times \mathcal{E}nv \to \mathcal{V}al & \\
\mathcal{E}[\![x]\!]\rho & \hat{=} & \rho(x) & \text{variables} \\
\mathcal{E}[\![a]\!]\rho & \hat{=} & a & \text{constantes} \\
\mathcal{E}[\![e_1 \,\mathbf{+}\, e_2]\!]\rho & \hat{=} & [\![e_1]\!]\rho + [\![e_2]\!]\rho & \text{addition, concaténation} \\
\mathcal{E}[\![e_1 * e_2]\!]\rho & \hat{=} & [\![e_1]\!]\rho \times [\![e_2]\!]\rho & \text{multiplication} \\
\mathcal{E}[\![e_1 \,/\, e_2]\!]\rho & \hat{=} & [\![e_1]\!]\rho / [\![e_2]\!]\rho & \text{division} \qquad \ldots
\end{array}
$$

Les tables d'évaluation de $+$, $\times$ et $/$ sont les suivantes, où concat est la concaténation de chaînes, $\lfloor x \rfloor$ la partie entière de x, $\neg$ la négation logique :

$\mathbf{+}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$n_2 \in \mathbb{Z}$	$s_2 \in \mathcal{S}tring$	autres
$\boxed{\text{W}}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{W}}$	$\boxed{\text{W}}$
$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{E}}$	$\boxed{\text{E}}$	$\boxed{\text{E}}$
$n_1 \in \mathbb{Z}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$n_1 + n_2$	$\boxed{\text{W}}$	$\boxed{\text{W}}$
$s_1 \in \mathcal{S}tring$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{W}}$	concat(s_1, s_2)	$\boxed{\text{W}}$
autres	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{W}}$	$\boxed{\text{W}}$

$*$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$n_2 \in \mathbb{Z}$	autres
$\boxed{\text{W}}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{W}}$
$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{E}}$	$\boxed{\text{E}}$
$n_1 \in \mathbb{Z}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$n_1 n_2$	$\boxed{\text{W}}$
autres	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{W}}$

$/$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$n_2 \in \mathbb{Z}$	autres
$\boxed{\text{W}}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{W}}$
$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{E}}$	$\boxed{\text{E}}$
$n_1 \in \mathbb{Z}$	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\begin{cases} \lfloor n_1/n_2 \rfloor & (n_2 \neq 0) \\ \boxed{\text{E}} & (n_2 = 0) \end{cases}$	$\boxed{\text{W}}$
autres	$\boxed{\text{W}}$	$\boxed{\text{E}}$	$\boxed{\text{W}}$	$\boxed{\text{W}}$

Pour relier cette sémantique concrète à la sémantique abstraite des signes, il nous faut donner une relation entre les signes — les valeurs *abstraites* — et les propriétés *concrètes* qu'elles signifient. Par exemple, $\geq \mathbf{0}$ devrait signifier “être positif ou nul”. Formellement, on définit une fonction γ dite de *concrétisation*, qui envoie chaque valeur abstraite $v^\sharp \in L^\sharp \hat{=} \{\geq \mathbf{0}, \leq \mathbf{0}, \top\}$ vers l'ensemble des valeurs qui vérifient la propriété correspondante. Ici, on définira donc :

$$
\begin{array}{rcl}
\gamma(\geq \mathbf{0}) & \hat{=} & \{n \in \mathbb{Z} \mid n \geq 0\} \cup \{\boxed{\text{E}}\} \\
\gamma(\leq \mathbf{0}) & \hat{=} & \{n \in \mathbb{Z} \mid n \leq 0\} \cup \{\boxed{\text{E}}\}
\end{array}
\qquad \gamma(\top) \hat{=} \mathcal{V}al \cup \{\boxed{\text{E}}, \boxed{\text{W}}\}
$$

On notera que $\top$ signifie “je ne sais pas” : toutes les valeurs, y compris les erreurs $\boxed{\text{E}}$ et $\boxed{\text{W}}$ sont possibles. Par contre, on voit que $\geq \mathbf{0}$ ne signifie pas “être un entier positif ou nul” mais “être un entier positif ou nul, ou bien une erreur à l'exécution ($\boxed{\text{E}}$), mais pas une erreur de type ($\boxed{\text{W}}$)”. C'est un choix qui, s'il semble arbitraire à première vue, est justifié par le problème de la ligne (3) : ainsi, le résultat de $1 \div 0$, $\boxed{\text{E}}$, est bien dans la concrétisation de $\geq \mathbf{0}$.

En général, une fois choisies une concrétisation γ et une fonction d'évaluation concrète $e, \rho \mapsto \mathcal{E}[\![e]\!]\rho$, une fonction d'évaluation abstraite $e, \rho^\sharp \mapsto \mathcal{E}[\![e]\!]^\sharp \rho^\sharp$ (dans l'exemple, `expr_sign`) est *correcte* si et seulement si toute valeur concrète de e est dans la concrétisation de sa valeur abstraite. Formellement :

Théorème de correction : si $\rho(x) \in \gamma(\rho^\sharp(x))$ pour tout $x \in \mathcal{V}ar$, alors $\mathcal{E}[\![e]\!]\rho \in \gamma(\mathcal{E}[\![e]\!]^\sharp \rho^\sharp)$ pour tout $e \in \mathcal{E}xpr$.

Le lecteur intéressé vérifiera que ce théorème est effectivement valide dans le cas des signes, avec le γ décrit plus haut.

On notera au passage que la ligne (2) est elle-même correcte. Selon notre définition de γ, $\top$ est bien un signe correct pour n'importe quelle chaîne : pour tout $s \in \mathcal{S}tring$, $s \in \gamma(\top)$. De plus, il se trouve que $\top$ est le signe *le plus précis* que l'on puisse donner à une chaîne s. En effet, les signes plus précis $\geq \mathbf{0}$ et $\leq \mathbf{0}$ sont tous les deux incorrects, car $s \notin \gamma(\geq \mathbf{0})$ et $s \notin \gamma(\leq \mathbf{0})$.

Ceci demande à définir la notion de précision des valeurs abstraites. Intuitivement, $\geq \mathbf{0}$ et $\leq \mathbf{0}$ sont tous les deux plus précis que $\top$, mais ni $\geq \mathbf{0}$ n'est plus précis que $\leq \mathbf{0}$ ni $\leq \mathbf{0}$ que $\geq \mathbf{0}$. On formalise ceci en définissant une relation d'ordre $\sqsubseteq^\sharp$ ("plus précis que"). Dans le cas des signes, $\geq \mathbf{0} \sqsubseteq^\sharp \top$, $\leq \mathbf{0} \sqsubseteq^\sharp \top$ et $\geq \mathbf{0}$ et $\leq \mathbf{0}$ sont incomparables. Cette notion d'ordre est correcte si, dès que $v_1^\sharp \sqsubseteq^\sharp v_2^\sharp$, alors la signification $\gamma(v_1^\sharp)$ de $v_1^\sharp$ est un ensemble de valeurs plus petit, pour l'inclusion, que $\gamma(v_2^\sharp)$. Cette propriété est la *monotonie* de γ :

$$v_1^\sharp \sqsubseteq^\sharp v_2^\sharp \Rightarrow \gamma(v_1^\sharp) \sqsubseteq^\flat \gamma(v_2^\sharp) \qquad [1]$$

Vouloir trouver le signe le plus précis possible mène naturellement à la définition d'une fonction d'abstraction α qui à toute propriété, c'est-à-dire tout ensemble S de valeurs et erreurs, associerait la valeur abstraite $v^\sharp$ la plus précise qui soit correcte. Notons que "correcte" signifie que tout élément x de S doit avoir la propriété symbolisée par $v^\sharp$, c'est-à-dire $x \in \gamma(v^\sharp)$. Formellement :

$$\alpha(S) \hat{=} \min_{\sqsubseteq^\sharp} \{v^\sharp \in L^\sharp \mid S \subseteq \gamma(v^\sharp)\} \qquad [2]$$

Le hic, c'est que le minimum $\min_{\sqsubseteq^\sharp} S^\sharp$ (la valeur la plus précise) d'un ensemble de valeurs abstraites $S^\sharp$ n'existe pas toujours. En particulier, c'est le problème que nous avions en ligne (1) : il n'y a pas de signe le plus précis qui représente le 0 correctement. Tous les signes possibles, $\geq \mathbf{0}$, $\leq \mathbf{0}$ et $\top$ sont en effet corrects (0 est dans les γ de tous), mais il n'y en a pas de plus précis que les autres.

Il est donc nécessaire de compléter notre ensemble de signes. Par exemple, on peut rajouter une valeur abstraite $\mathbf{0}$ telle que $\gamma(\mathbf{0}) \hat{=} \{0\} \cup \{\boxed{\mathrm{E}}\}$, ce que nous supposerons dans la suite. Alors $\alpha(\{0\}) = \mathbf{0}$.

En général, étant donnés deux ensembles ordonnés $(L^\sharp, \sqsubseteq^\sharp)$ et $(L^\flat, \sqsubseteq^\flat)$ (dans notre cas, l'ensemble des parties de $\mathcal{V}al \cup \{\boxed{\mathrm{E}}, \boxed{\mathrm{W}}\}$ ordonné par $\subseteq$), un couple de fonctions (α, γ), $\alpha : L^\flat \to L^\sharp$, $\gamma : L^\sharp \to L^\flat$ telles que les équations (1) et (2) sont satisfaites est appelée une *correspondance de Galois*.

Une fois une correspondance de Galois définie, l'interprétation abstraite d'une expression vient automatiquement. Par exemple, on va définir la valeur abstraite de e_1**+**e_2 comme étant la valeur abstraite la plus précise qui représente correctement toutes les sommes possibles de valeurs de e_1 et e_2. Formellement :

$$\begin{array}{rcll} v^\sharp, \ldots \in L^\sharp & \hat{=} & \{\mathbf{0}, \geq \mathbf{0}, \leq \mathbf{0}, \top\} & \text{valeurs abstraites} \\ \rho^\sharp, \ldots \in \mathcal{E}nv^\sharp & \hat{=} & \mathcal{V}ar \to L^\sharp & \text{environnements abstraits} \end{array}$$

$$\begin{array}{rcll} \mathcal{E}[\![\_]\!]^\sharp\_ & : & \mathcal{E}xpr \times \mathcal{E}nv^\sharp \to L^\sharp & \\ \mathcal{E}[\![x]\!]^\sharp\rho^\sharp & \hat{=} & \rho^\sharp(x) & \text{variables} \\ \mathcal{E}[\![a]\!]^\sharp\rho^\sharp & \hat{=} & \alpha(\{a\}) & \text{constantes} \\ \mathcal{E}[\![e_1 \mathbin{\mathbf{+}} e_2]\!]^\sharp\rho^\sharp & \hat{=} & \alpha(\{x+y \mid x \in \gamma(\mathcal{E}[\![e_1]\!]^\sharp\rho^\sharp), y \in \gamma(\mathcal{E}[\![e_2]\!]^\sharp\rho^\sharp\})) & \\ \mathcal{E}[\![e_1 * e_2]\!]^\sharp\rho^\sharp & \hat{=} & \alpha(\{x \times y \mid x \in \gamma(\mathcal{E}[\![e_1]\!]^\sharp\rho^\sharp), y \in \gamma(\mathcal{E}[\![e_2]\!]^\sharp\rho^\sharp\})) & \\ \mathcal{E}[\![e_1 \,/\, e_2]\!]^\sharp\rho^\sharp & \hat{=} & \alpha(\{x/y \mid x \in \gamma(\mathcal{E}[\![e_1]\!]^\sharp\rho^\sharp), y \in \gamma(\mathcal{E}[\![e_2]\!]^\sharp\rho^\sharp\})) & \ldots \end{array}$$

Nous invitons le lecteur à en déduire explicitement les tables de calcul des signes et à compléter le code de la fonction `expr_sign`. (En exercice, montrer que $\mathbf{0}$ multiplié par toute valeur abstraite $v^\sharp$ vaut $\mathbf{0}$, sauf quand $v^\sharp = \top$, et expliquer pourquoi.)

Le domaine $\{\top, \geq \mathbf{0}, \leq \mathbf{0}, \mathbf{0}\}$ permet des analyses relativement peu précises. Par exemple, on ne peut pas l'utiliser pour déduire qu'une expression est strictement positive, ou bien que c'est un booléen et non un entier. Pour corriger ce problème, il suffit d'ajouter de nouvelles valeurs abstraites, d'étendre la définition de γ tout en vérifiant que α est toujours bien défini par l'équation (2), et de recalculer les tables de calcul abstrait associées. Le lecteur intéressé pourra s'exercer sur les valeurs abstraites $\top, \geq \mathbf{0}, \leq \mathbf{0}, \mathbf{0}$, plus:

$v^\sharp$	$\gamma(v^\sharp)$	$v^\sharp$	$\gamma(v^\sharp)$
$> \mathbf{0}$	$\{n \in \mathbb{Z} \mid n > 0\} \cup \{\boxed{\text{E}}\}$	$< \mathbf{0}$	$\{n \in \mathbb{Z} \mid n < 0\} \cup \{\boxed{\text{E}}\}$
$\neq \mathbf{0}$	$\{n \in \mathbb{Z} \mid n \neq 0\} \cup \{\boxed{\text{E}}\}$	`err`	$\{\boxed{\text{E}}\}$
`int`	$\mathbb{Z} \cup \{\boxed{\text{E}}\}$	`bool`	$\mathbb{B} \cup \{\boxed{\text{E}}\}$
`string`	$\mathcal{S}tring \cup \{\boxed{\text{E}}\}$	$\perp$	$\emptyset$

(En exercice, que vaut la division de $\neq \mathbf{0}$ par $\neq \mathbf{0}$ dans ce nouveau domaine abstrait, en particulier pourquoi ne peut-elle pas valoir $\neq \mathbf{0}$?)

Un point à noter ici est que l'introduction des valeurs abstraites `int`, `bool` et `string` permet naturellement de faire une analyse de *types* des expressions, et l'on a le résultat important suivant, qui exprime que les expressions e bien typées (de valeur abstraite différente de $\top$) ne provoquent pas d'erreur de type à l'exécution (n'ont pas la valeur abstraite $\boxed{\text{W}}$) :

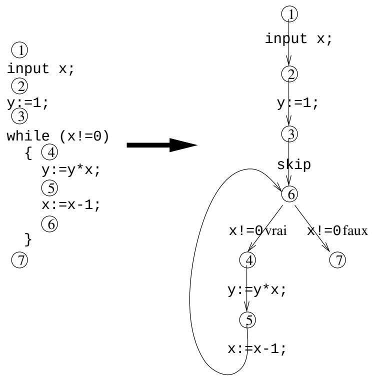
Théorème d'évaluation sans erreur de type : si $\rho(x) \in \gamma(\rho^\sharp(x))$ pour tout $x \in \mathcal{V}ar$ et $\mathcal{E}[\![e]\!]^\sharp\rho^\sharp \neq \top$, alors $\mathcal{E}[\![e]\!]\rho \neq \boxed{\text{W}}$.

En effet, par le théorème de correction si $\mathcal{E}[\![e]\!]\rho = \boxed{\text{W}}$ alors $\boxed{\text{W}} \in \gamma(\mathcal{E}[\![e]\!]^\sharp\rho^\sharp)$, mais la seule valeur abstraite $v^\sharp$ telle que $\boxed{\text{W}} \in \gamma(v^\sharp)$ est $\top$. Le *bytecode verifier* Java [LY96] est une application typique de ces techniques, qui utilise un domaine de types plus complexe dans le but de s'assurer qu'aucune erreur de type ne peut survenir à l'exécution. Les applets Java peuvent alors être exécutées sans faire de test de type à l'exécution, ce qui améliore d'autant leur efficacité.

1.4. *Approximation de points fixes et boucles*

L'utilisation de domaines ordonnés permet aussi de résoudre le point (4) soulevé par notre prototype `com_sign` : si $\rho^\sharp_1 \hat{=}$`com_sign rho c1` est l'environnement de signes retournés sur la branche `c1` du test, et $\rho^\sharp_2 \hat{=}$`com_sign rho c2` celui de la branche `c2`, alors l'environnement abstrait $\rho^\sharp$ pour `IF (e, c1, c2)` doit envoyer chaque variable x vers $\alpha(\gamma(\rho^\sharp_1(x)) \cup \gamma(\rho^\sharp_2(x)))$, la valeur abstraite la plus précise qui approche l'union des ensembles de valeurs possibles de x sur `c1` ($\gamma(\rho^\sharp_1(x))$) et sur `c2` ($\gamma(\rho^\sharp_2(x))$). Plus simplement, $\rho^\sharp(x)$ est le *supremum* $\rho^\sharp_1(x) \sqcup^\sharp \rho^\sharp_2(x)$ de $\rho^\sharp_1(x)$ et de $\rho^\sharp_2(x)$, c'est-à-dire la plus petite valeur abstraite supérieure ou égale à chaque. (Le lecteur est invité à calculer la table des supremums dans $L^\sharp \hat{=} \{\top, \geq \mathbf{0}, \leq \mathbf{0}, \mathbf{0}\}$; par exemple, que vaut $\geq \mathbf{0} \sqcup^\sharp \leq \mathbf{0}$?)

Restent à résoudre les points (5) et (6). L'approche standard pour résoudre le point (6), et donner des estimations des environnements à chaque point du programme, est de changer de représentation du programme à analyser. Nous représentons maintenant chaque programme comme un graphe orienté, le *graphe de contrôle*, dont les sommets sont appelés les *points de programmes* et se trouvent entre chaque instruction, et dont les arcs sont étiquetés par des *transitions*, qui sont à peu de choses près les instructions du programme. Par exemple, le graphe de contrôle du programme de gauche ci-dessous, où les points de programme ont été numérotés de 1 à 7, est dessiné en partie droite.



On notera que la boucle `while` est représentée par un circuit dans le graphe, et que le test `x!=0` est représenté par deux arcs, l'un exprimant la réussite du test (`x!=0` vrai?), l'autre son échec (`x!=0` faux?). En Caml, les transitions sont :

```
type etiq = SET of var * expr   | INPUT of var    | SKIP
   | TEST of expr * bool (* e vrai?, e faux? *);;
```

Cette représentation permet de parler de l'environnement abstrait $Env^\sharp(p)$ *au point de programme* p, $p \in \{1, \ldots, 7\}$. Intuitivement, $Env^\sharp(p)$ est *correct* si et seulement si, quelle que soit l'exécution concrète du programme, pour toute variable x, chaque fois que l'on passe par le point de programme p, alors la valeur de x est nécessairement dans $\gamma(Env^\sharp(p)(x))$. Une façon de calculer $Env^\sharp(p)$ est de simuler les exécutions concrètes en suivant le graphe de contrôle, en calculant dans le domaine abstrait $L^\sharp$ au lieu du domaine concret $L^\flat$.

Illustrons une réalisation possible en Caml de cette méthode de calcul. Nous allons maintenir une table de hachage $E^\sharp$ qui à chaque point de programme p pris comme index va associer $E^\sharp(p)$, notre estimation courante de $Env^\sharp(p)$. Initialement, $E^\sharp(p)$ est l'environnement qui à chaque variable associe l'élément minimum $\perp$ de $L^\sharp$ (l'information la plus précise possible, mais peut-être incorrecte). On écrit maintenant une fonction **cfg_sign** telle que **cfg_sign** $E^\sharp$ $\rho^\sharp$ **graphe** p met à jour $E^\sharp$, sachant que le graphe de contrôle est donné par **graphe**, et que l'on souhaite corriger $E^\sharp(p)$ en $E^\sharp(p) \sqcup^\sharp \rho^\sharp$ et propager les changements subséquents dans $E^\sharp$. En pseudo-code :

let rec cfg_sign $E^\sharp$ $\rho^\sharp$ graphe p =
   if $E^\sharp(p) = \rho^\sharp$
      then return;   (* résultat stable, le calcul a convergé. *)
   else ($E^\sharp(p)$:=$E^\sharp(p) \sqcup^{\sharp *} \rho^\sharp$;   (* mise à jour. *)
         pour tous q et *instr* tels que $p \xrightarrow{instr} q$ est un arc de graphe :
            cfg_sign $E^\sharp$ ($cond^\sharp[\![instr]\!]\rho^\sharp$) graphe q);;

où l'opération $\sqcup^{\sharp *}$ calcule le supremum de deux environnements abstraits, variable par variable : $(\rho_1^\sharp \sqcup^{\sharp *} \rho_2^\sharp)(x) \hat{=} \rho_1^\sharp(x) \sqcup^\sharp \rho_2^\sharp(x)$, et $cond^\sharp[\![instr]\!]\rho^\sharp$ calcule l'environnement abstrait résultant de l'exécution de la transition *instr* à partir d'un état où l'environnement abstrait est ρ :

$$\begin{array}{rcl}
cond^\sharp[\![x{:=}e]\!]\rho^\sharp & \hat{=} & \rho^\sharp[x \leftarrow \mathcal{E}[\![e]\!]^\sharp\rho^\sharp] \\
cond^\sharp[\![\mathtt{input}\,x]\!]\rho^\sharp & \hat{=} & \rho^\sharp[x \leftarrow \top] \\
cond^\sharp[\![e \text{ vrai?}]\!]\rho^\sharp & \hat{=} & \left\{\begin{array}{ll} \rho^\sharp & (\text{si } \mathtt{true} \in \gamma(\mathcal{E}[\![e]\!]^\sharp\rho^\sharp)) \\ \lambda x \cdot \perp & (\text{sinon}) \end{array}\right. \\
cond^\sharp[\![e \text{ faux?}]\!]\rho^\sharp & \hat{=} & \left\{\begin{array}{ll} \rho^\sharp & (\text{si } \mathtt{false} \in \gamma(\mathcal{E}[\![e]\!]^\sharp\rho^\sharp)) \\ \lambda x \cdot \perp & (\text{sinon}) \end{array}\right. \\
cond^\sharp[\![\mathtt{skip}]\!]\rho^\sharp & \hat{=} & \rho^\sharp
\end{array}$$

(En exercice, traduire la condition $\mathtt{true} \in \gamma(\mathcal{E}[\![e]\!]^\sharp\rho^\sharp)$ en une condition calculable. Dans le domaine des signes, cette condition est équivalente à $\mathcal{E}[\![e]\!]^\sharp\rho^\sharp = \top$; qu'en est-il dans le domaine étendu de la fin de la section 1.3 ? D'autre part, si le $E^\sharp$ calculé par **cfg_sign** retourne $E^\sharp(p)(x) = \perp$ pour un certain point de programme p, où l'on rappelle que $\gamma(\perp) = \emptyset$, que peut-on conclure ? On dira qu'un morceau de code est *mort* si aucune exécution ne permet d'y passer.)

La fonction `cfg_sign` calcule l'approximation abstraite des signes en chaque point de programme comme un plus petit point fixe. Ceci est justifié mathématiquement par le théorème suivant. Rappelons que la sémantique concrète d'un programme peut s'exprimer comme le plus petit point fixe $lfp_{\sqsubseteq^\flat}(f)$ d'une fonction monotone f. Ce théorème exprime que le plus petit point fixe de la fonction $\alpha \circ f \circ \gamma$, qui va du domaine abstrait $L^\sharp$ dans lui-même, est une approximation correcte du plus petit point fixe de f :

Théorème (Cousot) : si $f : L^\flat \to L^\flat$ est une fonction monotone, alors $lfp_{\sqsubseteq^\flat}(f) \sqsubseteq^\flat \gamma(lfp_{\sqsubseteq^\sharp}(\alpha \circ f \circ \gamma))$.

Plutôt que de prouver ce théorème ou de montrer précisément en quoi il justifie `cfg_sign` formellement, nous illustrons la façon dont `cfg_sign` propage les signes le long du graphe de contrôle en figure 1, où `cfg_sign` est appelé sur les arguments $E^\sharp \hat{=} \lambda p, x \cdot \bot$, $\rho^\sharp \hat{=} \lambda x \cdot \top$ (en entrée du programme, on ne sait rien sur les variables x, et $\top$ est donc l'information correcte la plus précise sur x), et $p \hat{=} 1$ (le point d'entrée du programme). Les points de programmes non décorés sont tels que $E^\sharp = \lambda p, x \cdot \bot$.

On notera que le calcul de point fixe effectué par `cfg_sign` résout le problème (6), mais aussi le problème (5). En effet, comme dans la figure 1, les valeurs abstraites associées aux variables à chaque point de programme ne peuvent que croître dans l'ordre $\sqsubseteq^\sharp$ (ne peuvent que devenir de moins en moins précises), mais cette croissance est forcée de s'arrêter : on peut faire croître les variables comme $\bot \to \mathbf{0} \to \geq \mathbf{0} \to \top$, ou comme $\bot \to \mathbf{0} \to \leq \mathbf{0} \to \top$ par exemple, mais toutes ces suites sont finies.

1.5. *Pour aller plus loin*

Nous espérons que les exemples, les explications et les morceaux de code ci-dessus auront su donner une idée des techniques utilisées aujourd'hui en analyse statique de programmes, et même convaincre le lecteur que ces techniques sont d'un niveau de complexité abordable. En particulier, il pourra utiliser, comprendre et même adapter l'analyseur générique de P. Cousot [Cou98]. Nous incitons d'autre part le lecteur intéressé à approfondir le sujet, que nous avons à peine effleuré. Notamment :

Comment garantir qu'un programme d'analyse statique termine dans le cas général ? Si le domaine des valeurs abstraites contient un domaine d'intervalles d'entiers $[m, n]$ par exemple (comme dans [Pol00], où il est utilisé par approcher les domaines de variation d'indices de tableaux, dans le but de détecter des accès illégaux possibles en-dehors des tableaux), l'itération de point fixe peut produire des chaînes infinies de valeurs : $[0, 0] \subseteq [0, 1] \subseteq [0, 2] \subseteq \ldots$, et le programme d'analyse statique ne termine plus. Ceci est réparé par l'usage

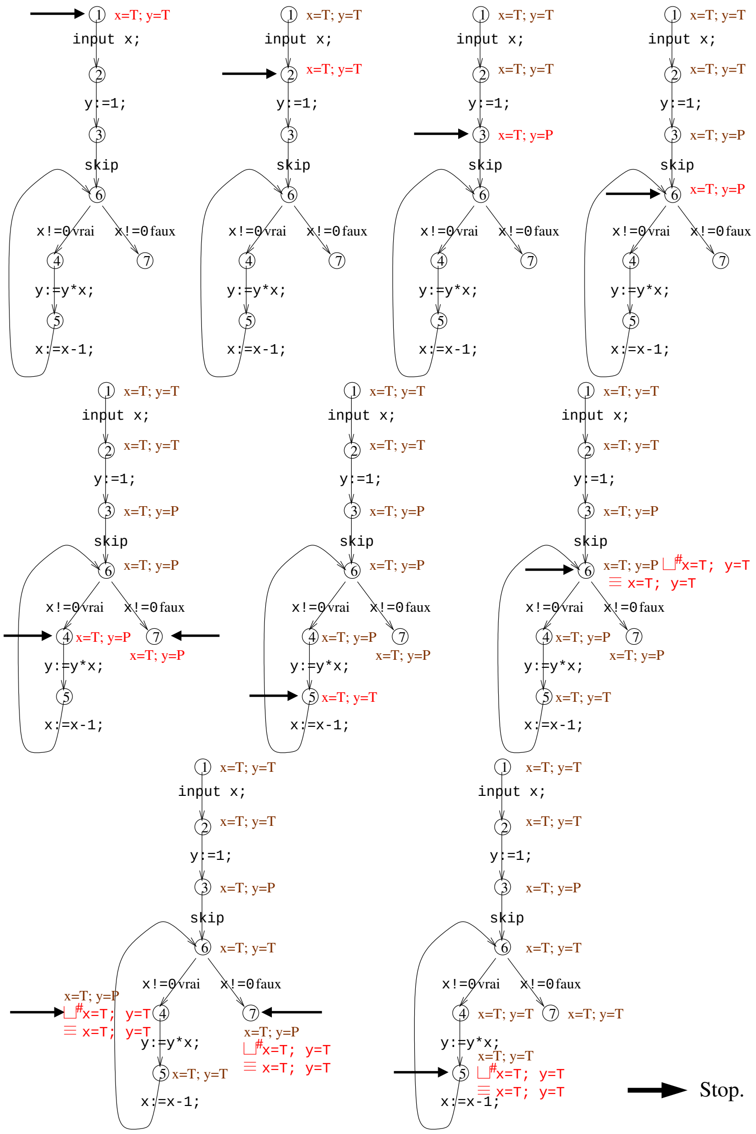


Figure 1. *Propagation des signes*

d'opérateurs d'*élargissement* (voir [CC92]), qui forcent la terminaison au prix d'une perte parfois drastique de précision. Une partie de cette précision peut être récupérée par une seconde passe utilisant des opérateurs de *rétrécissement*.

Comment calculer une combinaison de plusieurs sortes d'information ? Par exemple, notre exemple de calcul de signes et de types avec un calcul de variables non initialisées, de pointeurs nuls/non nuls, etc. ? Une solution courante est d'effectuer tous ces calculs dans autant d'analyseurs indépendants, mais ceci provoque des pertes d'informations inutiles. Une solution élégante est celle du *produit réduit*, voir [CC92, Gra92].

Comment récupérer de l'information à partir de la réussite ou du succès des tests ? L'analyseur `cfg_sign` appliqué sur le programme `if x=0 then skip else x`$=3$: retourne le signe $\top$ pour `x`. Pourtant si le test réussit, alors `x` vaut 0 par définition, donc est $\geq \mathbf{0}$, et s'il échoue, alors `x` vaut 3, qui est encore $\geq \mathbf{0}$. Une façon de déduire que dans la branche `then x` vaut 0 est d'effectuer une analyse *arrière* : au lieu de calculer un environnement abstrait après la transition en fonction de l'environnement abstrait avant, on fait calculer l'avant en fonction de l'après (si `x=0` réussit, alors `x` vaut 0). L'article [Gra92] donne des moyens intelligents pour atteindre ce but, en particulier.

Quelle direction pour la sémantique abstraite ? L'analyseur `cfg_sign` propage les valeurs abstraites du début à la fin du programme, en avant. Ceci permet de répondre à des questions de la forme : "connaissant les conditions initiales, que peut-on dire de `x` au point p ?". À l'opposé, on peut propager des valeurs abstraites en arrière, répondant ainsi à des questions de la forme : "sachant que la propriété P est vraie au point p, quelles étaient les conditions initiales ?" C'est utile notamment en *debugging abstrait* (voir [Bou93] par exemple), pour retrouver les causes possibles d'un bogue donné.

Dans quelle direction itérer ? C'est un point sutil : `cfg_sign` calcule les valeurs abstraites directement, mais on peut effectuer un analyseur en deux passes, la première calculant des *équations sémantiques* dont $Env^\sharp$ est la solution, la deuxième résolvant ces équations sémantiques. L'intérêt est que la résolution peut parcourir les équations à résoudre dans un ordre différent de l'ordre de leur génération. Ceci permet d'accélérer, parfois beaucoup, le calcul abstrait; voir par exemple [CH78]. On peut même résoudre ces équations par d'autres moyens, éventuellement non itératifs : c'est ce qui est fait dans l'inférence de types à la ML [DM82, Cou97]. L'inférence de types fournit souvent une méthode efficace et parfois facile à comprendre de déduction de propriétés de programmes. On citera entre autres [LG88, TJ92] où un système de types étendu est utilisé pour découvrir des expressions calculables en parallèle dans des programmes séquentiels, ou [RL98] qui permet d'assurer la sécurité d'environnements où viennent s'exécuter des morceaux de code mobile (des *applets*).

Et les relations entre variables ? Toute propriété trouvée par `cfg_sign` porte sur une variable, indépendamment des autres. Mais on peut aussi découvrir des relations entre variables par interprétation abstraite : ce sont les analyses *relationnelles*, comme celle de Karr [Kar78] qui découvre des relations affines, par exemple $\mathtt{x} + 2\mathtt{y} - 10\mathtt{z} = -1$, entre variables, ou celle, plus complexe et utilisée en parallélisation automatique de programmes scientifiques, des polyèdres convexes de Cousot et Halbwachs [CH78].

Que faire dans les langages à pointeurs ? Notre langage jouet n'avait pas de pointeur, comme en C, mais aussi Pascal, ML (références), ou Java (objets). En présence de pointeurs, les analyses sont plus difficiles. Par exemple, peut-on remplacer la dernière instruction de :

```
x->a = 3;  y->a = 5;  z = x->a;
```

par l'instruction `z = 3` ? On le peut uniquement si on peut assurer que `&x->a`$\neq$`&y->a`, autrement dit que `x->a` et `y->a` ne peuvent pas être des *alias*. La détection d'alias possibles de façon suffisamment précise est difficile, on consultera notamment [Deu94, SRW98].

Et en présence d'appels de procédures/fonctions/méthodes ? Les techniques que nous avons vues s'appliquent encore : on attribue un point de programme à chaque instruction de chaque procédure. Les appels récursifs sont traités comme des boucles. Mais ce n'est souvent pas assez précis, tous les appels à une même fonction se retrouvant amalgamés en une seule information abstraite. Dans certains cas, une analyse symbolique peut aider [SRH95]; le *partitionnement*, qui consiste à faire comme si on avait plusieurs copies de la même fonction à analyser, permet d'affiner les analyses, voir [Bou95].

Quelles autres propriétés ? Nous n'avons examiné que des propriétés de *sûreté*, en particulier, des *invariants*, c'est-à-dire des propriétés de la forme "la propriété P est toujours vraie", par exemple "au point p, `x` est toujours $\geq \mathbf{0}$". Mais on peut s'intéresser à d'autres propriétés : "le programme P *termine* toujours/parfois", "dans toute exécution, P finit toujours par écrire sur `stdout`", "dans toute exécution, P écrit sur `stdout` infiniment souvent". Ce sont des *propriétés temporelles*, que l'on peut traiter par des méthodes de *model-checking*. L'*analyse dataflow*, et en particulier la détection de code mort, de variables vivantes, de sous-expressions communes, ou d'invariants de boucle par exemple sont des cas particuliers [SS98] (voir chapitre ??).

Et encore ? L'interprétation abstraite est encore utilisée pour détecter des problèmes liés à l'utilisation du parallélisme de façon non contrôlée (interblocages, non-déterminisme, etc. [LMR+98, Pol00]), ou bien des contraintes d'intégrité dans les bases de données [BDS95], ou bien encore la sécurité des protocoles cryptographiques [GL00], entre autres. Le lecteur intéressé par les débuts de l'interprétation abstraite pourra se référer à [CC76].

Bibliographie

[Abs00] AbsInt Angewandte Informatik, `http://www.absint.com/`, 2000.

[BDS95] Véronique Benzaken, Anne Doucet, and Xavier Schaefer. Integrity constraint checking optimization based on abstract databases generation and program analysis. *Journal de l'ingénierie des systèmes d'information*, 1(3), 1995.

[Bou93] François Bourdoncle. Abstract debugging of higher-order imperative languages. *ACM SIGPLAN Notices*, 28(6):46–55, 1993.

[Bou95] François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1995.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the second international symposium on Programming*, pages 106–130. Dunod, 1976. Disponible en `http://www.di.ens.fr/~cousot/COUSOTpapers/ISOP76.shtml`.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 1992. Disponible en `http://www.di.ens.fr/~cousot/COUSOTpapers.shtml`.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.

[Cou97] Patrick Cousot. Types as abstract interpretations. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 316–331. ACM Press, January 1997.

[Cou98] Patrick Cousot. Marktoberdorf'98 generic abstract interpreter. `http://www.dmi.ens.fr/~cousot/Marktoberdorf98.shtml`, 1998.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM Conference on Programming Language Design and Implementation (PLDI'94)*, pages 230–241. ACM Press, 1994.

[DM82] L. Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM, ACM Press, January 1982.

[GL00] Jean Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *Workshop on formal methods in parallel programming, theory and applications (FMPPTA'2000)*, pages 977–984. Springer Verlag LNCS 1800, 2000.

[Gra92] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. In *12th Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'92)*, pages 68–79. Springer Verlag Lecture Notes in Computer Science 652, 1992.

[Kar78] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1978.

[LG88] J. M. Lucassen and Dave M. Gifford. Polymorphic effect systems. In *15th ACM Symposium on Principles of Programming Languages (POPL'88)*, page 1988. ACM Press, 1988.

[LMR+98] P. Lacan, J.N. Montfort, Le Vinh Quy Ribal, Alain Deutsch, and Georges Gonthier. The software reliability verification process: The ARIANE 5 example. In *Proceedings DASIA'98 (DAta Systems In Aerospace)*, volume SP-422. ESA Publications, Athènes, Grèce, May 1998.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.

[Pol00] PolySpace technologies, `http://www.polyspace.com/`, 2000.

[RL98] François Rouaix and Xavier Leroy. Security properties of typed applets. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, 1998.

[SRH95] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Colloquium on Formal Approaches in Software Engineering (FASE'95)*, pages 651–665. Springer Verlag LNCS 915, 1995. Disponible en `http://www.cs.wisc.edu/wpis/papers/tapsoft95.ps`.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhardt Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998. Voir `http://www.cs.wisc.edu/~reps/`.

[SS98] David A. Schmidt and Bernhard Steffen. Data-flow analysis as model checking of abstract interpretations. In *5th Static Analysis Symposium (SAS'98)*. Springer Verlag LNCS 1503, 1998.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), 1992.