Analyse statique par interprétation abstraite : application à la détection de dépassement de tampon *

Xavier Allamigeon[1,2] et Charles Hymans[1]

[1] EADS Innovation Works, SE/CS – Suresnes, France
[2] CEA-LIST MeASI – Gif-sur-Yvette, France
firstname.lastname@eads.net

Résumé De nombreuses failles de sécurité sont rendues possibles par la présence de bogues dans les logiciels. Le dépassement de tampon (*buffer overflow*) est l'exemple typique de bogue qui permet la prise de contrôle d'une machine. Concevoir des logiciels de la taille et complexité actuelles, et exempts d'erreurs, relève de l'utopie. Nous présentons ici une analyse statique par interprétation abstraite [10] pour la sécurité : sans exécuter le logiciel, par inspection de code uniquement, elle permet d'assurer l'absence de dépassements de tampon dans un programme.

1 Introduction

Depuis la fabrication des premiers circuits intégrés, et soutenu par la tendance régulière de la loi de Moore, le logiciel n'a cessé de se propager dans les différents domaines technologiques. Aujourd'hui, et presque à notre insu, le logiciel revêt une importance considérable dans nos vies. Il assure des tâches aussi variées que la surveillance d'installations nucléaires, les commandes de vol électriques (*fly-by-wire*) des avions, les systèmes de freinage des voitures, et se retrouve dans les téléphones portables, le contrôle d'imagerie médicale, *etc.*

De plus, dans chaque domaine, la taille du code a tendance à croître de façon phénoménale. L'évolution de la taille des systèmes d'exploitation de Microsoft, reproduite à la figure 1, en est une excellente illustration. De même, un Boeing 777 transporte quatre millions de lignes de codes [40] et une voiture récente trente-cinq millions [30]. Bien qu'il réalise des fonctions de plus en plus complexes, et peut-être parce qu'il est intangible, on a tendance à sous-estimer les risques occasionnés par le logiciel. Ainsi, dans le cas de logiciel embarqué, il suffit d'une simple erreur de programmation pour rendre un produit inopérant, ou pire encore vulnérable à une exploitation malveillante.

Malgré tous les efforts de programmation, l'introduction malencontreuse d'erreurs est inévitable lors de l'écriture de logiciels de plusieurs millions de lignes de code. Selon les projets et les processus de développement, on peut compter de 0.5 à 5 erreurs pour cent lignes écrites. Une partie de ces erreurs est éliminée

* Ce travail a été réalisé avec le soutien du projet Usine Logicielle du pôle System@tic Paris-Région.

Année	Système d'exploitation	Lignes de code (en million)
1993	Windows NT 3.1	6
1994	Windows NT 3.5	10
1996	Windows NT 4.0	16
2000	Windows 2000	29
2001	Windows XP	40
2005	Windows Vista Beta 2	50

Fig. 1. Evolution de la taille du code de Microsoft Windows [47].

par une phase de relecture et de tests. Bien sûr, ces méthodes ne sont pas exhaustives. On estime que seuls 70% des bogues peuvent être découverts par une bonne relecture [24]. Les tests, eux, ne couvrent qu'une part de l'ensemble des comportements possibles des programmes (50% selon [24]). Après ces étapes de vérification, il n'y a donc aucune certitude quant à l'absence d'erreurs dans les logiciels, ce qui est inacceptable pour les codes critiques.

L'*analyse statique* est une technique permettant, sans exécuter le logiciel (par une inspection de son code source), de détecter les erreurs qu'il pourrait provoquer. Ils existent de nombreux outils fondés sur cette technique (voir la partie 6). Certains d'entre eux reposent sur l'*interprétation abstraite*, théorie introduite dans [10] et permettant de sur-approximer l'ensemble de *tous* comportements possibles d'un programme. Les analyses par interprétation abstraite sont alors dites *correctes* (ou *sûres*) : elles permettent de montrer formellement l'absence d'erreurs dans un logiciel.[3] Néanmoins, si les sur-approximations réalisées ne sont pas assez précises, une *fausse alarme* (ou *faux positif*) peut être signalée : l'analyse ne parvient pas dans ce cas à montrer l'absence d'erreur, alors que le programme n'en contient pas. L'enjeu est donc de construire des analyses suffisamment précises, mais aussi entièrement automatiques et assez performantes pour analyser des logiciels de plusieurs millions de lignes de code.

Dans cet article, nous construisons une analyse statique par interprétation abstraite permettant de montrer l'absence de dépassements de tampon dans un programme. Nous appliquerons en particulier l'analyse au programme de la figure 2, qui sera utilisé comme illustration au cours de l'article. La partie 2 présente tout d'abord le langage sur lequel porte l'analyse. Nous formalisons ensuite la sémantique de ce langage, c'est-à-dire la description mathématique du comportement de chaque construction (partie 3). Puis, nous présentons le principe de l'interprétation abstraite et la nature de la sur-approximation réalisée sur la sémantique (partie 4). Enfin, nous proposons des extensions possibles de l'analyse (partie 5), et décrivons des travaux relatifs à l'analyse statique en général (partie 6).

[3] En d'autres termes, toutes les erreurs que le programme pourrait produire sont détectées : il n'y a pas de *faux négatifs*.

```
void readbuf(unsigned char* t, unsigned int n) {
    unsigned int i,sz;
    sz = (unsigned int) getchar();
    if (sz > n)
       sz = n;
    i = 0;
    while (i < sz) {
       *(t+i) = (unsigned char) getchar();
       i++;
    }
}

unsigned char* receive(unsigned int n) {
    assert (n>0);
    unsigned char* t = (unsigned char*)malloc(n);
    readbuf(t,n);
    return t;
}
```

Fig. 2. Un programme en langage C à analyser. La fonction `receive` s'assure tout d'abord que la valeur de `n` est strictement positive, puis alloue un tampon `t` de `n` caractères dans le tas. Elle appelle ensuite la fonction `readbuf` afin de collecter dans ce tampon des données provenant d'une source extérieure, par des appels à une fonction `getchar`. Celle-ci lit par exemple l'entrée standard ou redirige des données provenant du réseau. Ce programme ne provoque pas d'erreur car la variable `i` a toujours une valeur plus petite que celle de `n` lors du déréférencement `*(t+i)`.

2 Langage et syntaxe

Analyser un programme écrit dans un langage complexe, comme C, est une tâche particulièrement difficile du fait de la richesse du langage. Aussi, pour des raisons de concision, nous construisons une analyse pour un langage qui peut être vu comme une forme « noyau » du C : il possède certaines caractéristiques de C (et d'autres langages impératifs), mais ses constructions sont beaucoup plus simples (et n'expriment qu'un sous-ensemble de C). Ce langage et son analyse peuvent être néanmoins enrichis de nombreuses manières, nous renvoyons le lecteur à la partie 5 pour plus de détails.

La traduction de programmes écrits en C vers ce langage noyau n'est pas détaillée ici. Néanmoins, il existe des formes intermédiaires pour C [37,29] à partir desquelles cette traduction serait aisée.

Dans la suite, nous présentons les principes du langage analysé puis sa syntaxe.

2.1 Principes du langage

Le seul type entier considéré dans le langage est **unsigned char** (abrégé en **uchar**), et les seuls pointeurs possibles sont de type **uchar**$*$. De cette manière,

un seul niveau de déréférencement est autorisé. Les conversions de type sont par ailleurs interdites. Nous supposons donc que le type de chaque variable est connu, et que l'ensemble des variables de type **uchar** est disjoint de celui des variables de type pointeur. Nous notons d'ailleurs Vars l'ensemble (fini) des variables, CharVars et PtrVars les ensembles (disjoints) des variables de type **uchar** et **uchar**$*$ respectivement.

Nous supposons l'existence d'une fonction getuchar renvoyant à chaque appel un caractère non-signé. Son comportement est analogue à celui de `getchar` dans le langage C : son résultat est lu à partir d'une source extérieure (entrée standard, fichier, réseau, *etc*).

Comme pour le langage C, la mémoire consiste en deux parties disjointes : la pile et le tas. La première contient la mémoire allouée de manière statique (par déclaration), la deuxième celle allouée de manière dynamique (par un appel à la fonction malloc). Nous supposerons que les valeurs stockées dans le tas sont toutes de type **uchar**. Nous considérons donc le tas comme une zone mémoire dans laquelle nous pouvons stocker des données comme des chaînes de caractères. L'allocation statique ou dynamique de la mémoire ne provoque pas d'initialisation des zones allouées (contrairement à la déclaration de variables globales avec la plupart des compilateurs en C).

Seuls les pointeurs vers une adresse du tas sont autorisés. La pile ne peut donc pas être modifiée par l'intermédiaire des pointeurs. Ceci est assuré par l'absence de l'opérateur & dans le langage. Pour des raisons de simplicité, nous excluons la valeur **null** pour les pointeurs.

Enfin, un programme est dépourvu de définitions ou d'appels de fonctions (hormis malloc et getuchar). Ses variables ont toutes la même portée, et sont déclarées au début du programme.

2.2 Syntaxe du langage

La syntaxe du langage est définie à la figure 3. Un programme consiste en une suite de commandes. Pour plus de lisibilité, nous avons choisi de ne pas faire apparaître la déclaration des variables : ces dernières sont donc déclarées implicitement avant la première commande du programme.

Les manipulations de la mémoire $instr_{stack}$ et $instr_{heap}$ seront appelées *instructions*. Chaque occurrence de la fonction malloc est décorée par un symbole α, appelé *site d'allocation*.[4] Dans un programme, les symboles de site d'allocation sont deux à deux distincts. Ils forment un ensemble (fini) noté Alloc.

Exemple 1 La figure 4 présente le programme $P_{receive}$, une possible traduction du programme donné en introduction dans la syntaxe de notre langage noyau. L'instruction `assert` du code initial n'est plus nécessaire, car comme nous le verrons la section 3.3, l'appel à la fonction $\mathsf{malloc}_\alpha(n)$ suppose déjà que l'entier n a une valeur strictement positive. Nous nous appuierons sur le code de $P_{receive}$ pour de nombreux exemples dans la suite de l'article. ❑

[4] Nous verrons dans la section 3.2 l'utilité d'un tel symbole.

$$
\begin{array}{rcll}
cmd & ::= & instr_{stack} & \text{instruction dans la pile} \\
& | & instr_{heap} & \text{instruction dans le tas} \\
& | & cmd_1; cmd_2 & \text{suite de commandes} \\
& | & \mathsf{while}\ cond\ \{\ cmd\ \} & \text{boucle} \\
& | & \mathsf{if}\ cond\ \{\ cmd_1\ \}\ \mathsf{else}\ \{\ cmd_2\ \} & \text{conditionnelle} \\
\\
instr_{stack} & ::= & x = e & \text{affectation arithmétique} \\
& | & x = \mathsf{getuchar}() & \text{lecture d'un caractère aléatoire} \\
& | & p = q + e & \text{affectation de pointeur} \\
& | & p = \mathsf{malloc}_\alpha(e) & \text{allocation dans le tas} \\
\\
instr_{heap} & ::= & *p = e & \text{affectation dans le tas} \\
\\
cond & ::= & e\ (\leq |\ ==)\ 0 & \text{comparaison à 0} \\
& | & !cond & \text{négation} \\
\\
e & ::= & c & \text{constante caractère} \\
& | & x & \text{variable de type } \textbf{uchar} \\
& | & \mathsf{op}(e_1, e_2) & \text{opération binaire}
\end{array}
$$

Fig. 3. Syntaxe du langage. (Le symbole p désigne une variable de type pointeur, op l'addition ou la soustraction sur les caractères.)

```
1 :   t = malloc_α(n);
2 :   sz = getuchar();
3 :   if (!(sz − n ≤ 0))
4 :     sz = n;
5 :   i = 0;
6 :   p = t;
7 :   while (i − sz + 1 ≤ 0) {
8 :     tmp = getuchar();
9 :     *p = tmp;
10 :    p = p + 1;
11 :    i = i + 1;
12 :  }
```

Fig. 4. Le programme $P_{receive}$: une traduction du programme donné à la figure 2. (Les variables i, n, sz et tmp sont de type **uchar**, p et t de type pointeur.)

3 Sémantique du langage

Dans cette partie, nous présentons la sémantique du langage introduit dans la partie 2. Nous définissons un modèle mathématique permettant de représenter les états de la mémoire (section 3.2), le comportement de chaque instruction (section 3.3) puis celui d'un programme (section 3.4). Enfin, nous formalisons les propriétés à vérifier pour qu'un programme donné ne produise pas d'erreurs à l'exécution (section 3.5).

3.1 Quelques prérequis

Avant de donner la sémantique exacte du langage de programmation, nous introduisons quelques notions et notations nécessaires pour la suite.

Graphe de flot de contrôle Nous supposons que tout programme P écrit dans le langage défini en partie 2 dispose d'un *graphe de flot de contrôle*, décrivant l'agencement des instructions et des conditions sous forme de graphe. Ce graphe orienté a pour noeuds les points de contrôle[5] du programme, et il existe un arc de i vers j s'il est possible de passer du point i au point j par l'exécution d'une instruction ou par l'application d'une condition. Nous étiquetons alors l'arc par l'instruction ou la condition correspondante.

Exemple 2 La figure 5 présente le graphe de flot de contrôle du programme $P_{receive}$ donné à l'exemple 1. ❑

Le graphe de flot de contrôle peut être construit de manière statique à partir du code du programme. Cette construction est classique, c'est pourquoi nous ne la détaillons pas.

Etant donné un programme P, nous notons $\langle i, stmt, j\rangle$ le fait qu'il existe un arc entre i et j étiqueté par *stmt* dans le graphe de flot de contrôle de P. Nous faisons l'hypothèse que le programme P possède un point de contrôle entrant, noté $entry(P)$.[6] Nous supposons que $entry(P)$ n'est atteint par aucun arc entrant.

Transitions et règles d'inférence Dans ce qui suit, nous allons décrire la sémantique du programme par une relation notée $\rightarrow$, dite *relation de transition*. Plus précisément, nous aurons $\sigma_1 \rightarrow \sigma_2$ lorsque par l'exécution d'une instruction ou l'application d'une condition, la machine peut passer de l'état σ_1 à l'état σ_2.

La relation $\rightarrow$ sera définie grâce à un ensemble de *règles d'inférence*. Chaque règle est de la forme :

$$\frac{C_1 \quad \ldots \quad C_n}{D}$$

où les C_i sont appelés *prémisses*, et D la *conclusion*. La signification d'une telle règle est la suivante : « si les C_i sont vrais, alors D est vrai ».

[5] Dans le cas où il n'y qu'une instruction par ligne, ces points de contrôle peuvent être assimilés aux numéros de ligne.

[6] En C, ce point correspond à la fonction *main*.

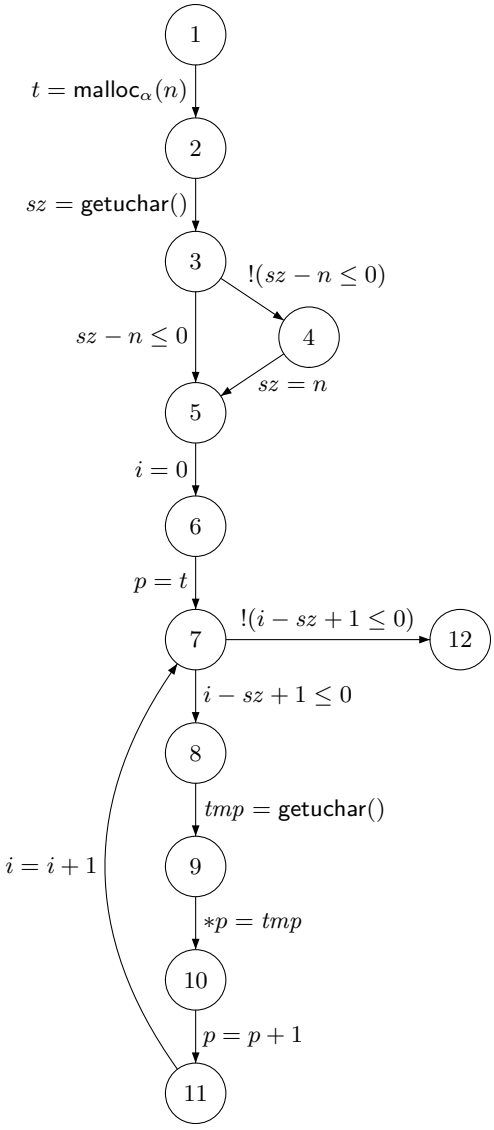


Fig. 5. Graphe de flot de contrôle du programme $P_{receive}$.

3.2 Modelisation des états de la mémoire

La mémoire est décrite par deux éléments, s et h, représentant respectivement la pile et le tas. L'état de la mémoire est alors noté $s \dagger h$. L'ensemble des états de la mémoire est noté Mem. Les choix faits pour la formalisation des états de la mémoire sont développés dans les parties suivantes, et récapitulés dans le tableau donné en figure 6.

Modélisation de la pile Une pile s associe à chaque variable du programme une valeur dont la nature dépend de son type (caractère ou pointeur).

On assimile l'ensemble des caractères à $\mathbb{N}$ (ensemble des entiers naturels). Cette simplification est acceptable si l'on suppose qu'il n'y a pas de dépassements de capacité dans les calculs arithmétiques sur les caractères. Cela nous permet de simplifier le formalisme.

La valeur d'un pointeur est soit la valeur spéciale ω, signifiant que le pointeur n'est pas initialisé, soit une adresse en mémoire, dont l'ensemble est noté Addr.[7] Ainsi, s est une application de Vars dans $\mathbb{N} \cup (\{\omega\} \cup \mathsf{Addr})$. L'ensemble des piles possibles est noté Stack.

Modélisation du tas De même, un tas h associe les adresses allouées dynamiquement à des caractères. Nous le modélisons donc comme une fonction partiellement définie sur Addr, à valeur dans $\mathbb{N}$. Alors que s est définie sur Vars tout entier, l'ensemble de définition de h n'est qu'un sous-ensemble de Addr : le sous-ensemble des adresses allouées dans le tas. Nous notons $\mathsf{dom}(h)$ le domaine de définition de h. D'ailleurs, l'écriture dans une adresse qui n'est pas définie dans h correspond à un dépassement de tampon, comme nous le verrons dans la section 3.5. Nous notons Heap l'ensemble des tas.

Modélisation des adresses Plutôt que de donner des valeurs entières aux adresses (comme c'est le cas sur une machine réelle), nous préférons modéliser les adresses de façon plus abstraite. En effet, la mémoire allouée dans le tas à l'utilisateur est une suite de blocs contigus. Dans notre formalisme, chacun de ces blocs est créé par un appel à une fonction malloc.

Afin d'assurer l'absence de dépassement de tampon, il suffit de montrer qu'aucune écriture en mémoire n'est réalisée en dehors de ces blocs. En particulier, il n'est pas nécessaire de connaître l'agencement exact de ces blocs les uns par rapport aux autres, si ce n'est de savoir qu'ils ne se chevauchent pas. Il est néanmoins essentiel que l'arithmétique des pointeurs ait du sens sur un bloc de mémoire donné.[8] C'est la raison pour laquelle nous décrivons chaque adresse a par un couple (l_α, o), où l_α est appelée *location*, et o est un entier positif. La location l_α correspond à un bloc alloué par la fonction malloc_α au site d'allocation α. C'est l'unique information que nous avons sur le bloc de mémoire en

[7] Naturellement, $\omega \notin \mathsf{Addr}$.

[8] C'est d'ailleurs la seule arithmétique dont le comportement soit défini dans la norme Ansi C [23, section 6.5.6, paragraphe 8].

$\mathsf{States} \stackrel{def}{=} \mathsf{Stack} \times \mathsf{Heap}$	$\mathsf{Stack} \stackrel{def}{=} (\mathbb{N} \cup \{\omega\} \cup \mathsf{Addr})^{\mathsf{Vars}}$
$\sigma = s \natural h$ vérifiant (1)	s application de Vars dans $(\mathbb{N} \cup \{\omega\} \cup \mathsf{Addr})$
$\mathsf{Heap} \stackrel{def}{=} \mathbb{N}^{(\mathsf{Addr})}$	$\mathsf{Addr} \stackrel{def}{=} \mathsf{Loc} \times \mathbb{N}$
h application partielle de Addr dans $\mathbb{N}$	$a = (l_\alpha, o)$

Fig. 6. Récapitulatif des choix faits pour le modèle de la mémoire. (Pour chaque ensemble, la forme des éléments est précisée en dessous.)

question. En particulier, nous ne connaissons pas exactement la position de ce bloc dans la mémoire de la machine. L'entier o est la composante de l'adresse qui rend possible l'arithmétique des pointeurs. Par exemple, l'adresse précédant (l_α, o) est $(l_\alpha, o-1)$, et celle la suivant est $(l_\alpha, o+1)$. Autrement dit, o est le *décalage* par rapport à l'adresse de début du bloc, $(l_\alpha, 0)$.

Nous noterons Loc l'ensemble des locations. Ainsi, l'ensemble des adresses peut être défini par $\mathsf{Addr} \stackrel{def}{=} \mathsf{Loc} \times \mathbb{N}$. Enfin, nous nous permettrons l'abus de notation $l_\alpha \in \mathsf{dom}(h)$ pour caractériser le fait qu'il existe $o \in \mathbb{N}$ tel que $(l_\alpha, o) \in \mathsf{dom}(h)$.

Etats de la mémoire bien formés Etant donnés une pile s et un tas h, nous imposons un invariant supplémentaire pour que $s \natural h$ soit un état de la mémoire valide :

$$\forall l_\alpha \in \mathsf{Alloc}. \forall p \in \mathsf{PtrVars}. \text{ si } s(p) = (l_\alpha, \cdot), \text{ alors } l_\alpha \in \mathsf{dom}(h) \tag{1}$$

Autrement dit, tout pointeur initialisé doit pointer vers un bloc existant en mémoire. Néanmoins, ce n'est pas pour autant que ce pointeur pointe bien sur une adresse située entre le début et la fin du bloc en question. Un dépassement de tampon est donc toujours possible.

Un tel invariant est acceptable dans la mesure où, comme nous le verrons dans la partie 3, la seule manière d'introduire une nouvelle location l_α dans l'ensemble des valeurs des pointeurs est d'appeler la fonction $\mathsf{malloc}_\alpha(\cdot)$, laquelle alloue en mémoire le bloc correspondant. De plus, les autres instructions conservent l'invariant.

Exemple 3 En reprenant le programme $P_{receive}$ donné à l'exemple 1, un état de la mémoire possible au point de contrôle 12 est $s_f \natural h_f$, où :

$$s_f : \begin{cases} i \mapsto 6 & p \mapsto (\lambda_\alpha, 6) \\ n \mapsto 8 & t \mapsto (\lambda_\alpha, 0) \\ sz \mapsto 6 & \\ tmp \mapsto 0 & \end{cases} \qquad h_f : \begin{cases} (\lambda_\alpha, 0) \mapsto 72 & (\lambda_\alpha, 4) \mapsto 111 \\ (\lambda_\alpha, 1) \mapsto 101 & (\lambda_\alpha, 5) \mapsto 0 \\ (\lambda_\alpha, 2) \mapsto 108 & (\lambda_\alpha, 6) \mapsto 179 \\ (\lambda_\alpha, 3) \mapsto 108 & (\lambda_\alpha, 7) \mapsto 23 \end{cases}, \tag{2}$$

avec $\lambda_\alpha \in \mathsf{Loc}$. Dans l'exécution qui a mené à cet état, la variable n vaut 8, c'est-à-dire que chaque message reçu doit avoir au plus 8 caractères. Ici, la taille du message reçu est égal à 6, et correspond à la chaîne de caractères `"Hello"` (avec un caractère nul de fin de chaîne). Le reste du bloc a un contenu quelconque, comme c'est effectivement le cas après un appel à malloc.

Dans la partie suivante, nous définissons précisément la sémantique du langage, et nous verrons que l'état ci-dessus est bien un état possible de la mémoire après l'exécution du programme. ❑

3.3 Sémantique opérationnelle

Etat de sémantique L'état σ de sémantique décrit l'état courant de la machine. Dans notre cas, il est constitué du point de contrôle courant i et d'un état de la mémoire $s \wr h$. Si Ctrl est l'ensemble des points de contrôle du programme, l'ensemble des états de sémantique est donc $\mathsf{States} \stackrel{def}{=} \mathsf{Ctrl} \times \mathsf{Mem}$.

Evaluation des expressions Nous définissons tout d'abord l'évaluation d'une expression e (de type **uchar**) dans un état de mémoire $s \wr h$ donné. Cette évaluation renvoie une valeur $v \in \mathbb{N}$, ce que nous notons $s \wr h \vdash e \Rightarrow v$. L'évaluation est définie par induction sur la forme de l'expression e :

$$\frac{c \in \mathbb{N}}{s \wr h \vdash c \Rightarrow c} \tag{3}$$

$$\frac{x \in \mathsf{CharVars}}{s \wr h \vdash x \Rightarrow s(x)} \tag{4}$$

$$\frac{s \wr h \vdash e_1 \Rightarrow v_1 \qquad s \wr h \vdash e_2 \Rightarrow v_2 \qquad \mathsf{op}(v_1, v_2) = v}{s \wr h \vdash \mathsf{op}(e_1, e_2) \Rightarrow v} \tag{5}$$

Sémantique des instructions La sémantique des instructions est donnée par la règle suivante : si *instr* est une instruction,

$$\frac{\langle i, \mathit{instr}, j\rangle \qquad s \wr h \vdash \mathit{instr} : s' \wr h'}{(i, s \wr h) \rightarrow (j, s' \wr h')} \tag{6}$$

où la relation $s \wr h \vdash \mathit{instr} : s' \wr h'$, définie à la figure 7, décrit l'effet de bord de l'instruction *instr* sur la mémoire. L'interprétation de ces règles est la suivante :

- l'affectation dans la pile $x = e$ consiste à remplacer dans s la valeur de x par celle de e.
- l'instruction $x =$ getuchar() est quant à elle non-déterministe : la fonction getuchar() peut en effet s'évaluer vers n'importe quel caractère c. Ceci correspond à un modèle prenant en compte tous les comportements possibles, y compris ceux hostiles, de la source extérieure.
- l'affectation de pointeur $p = q + e$ remplace dans s la valeur de p par celle de q décalée de e cases. Si q n'est pas initialisé, alors p prend la valeur ω.

$$\frac{s \mid h \vdash e \Rightarrow v}{s \mid h \vdash x = e : s[x \mapsto v] \mid h} \qquad \frac{c \in \mathbb{N}}{s \mid h \vdash x = \mathsf{getuchar}() : s[x \mapsto c] \mid h}$$

$$\frac{s(q) = (l_\alpha, o) \qquad s \mid h \vdash e \Rightarrow v}{s \mid h \vdash p = q + e : s[p \mapsto (l_\alpha, o + v)] \mid h} \qquad \frac{s(q) = \omega}{s \mid h \vdash p = q + e : s[p \mapsto \omega] \mid h}$$

$$\frac{s \mid h \vdash e \Rightarrow n \qquad n > 0 \qquad l_\alpha \notin \mathsf{dom}(h) \qquad w_0, \ldots, w_{n-1} \in \mathbb{N}}{s \mid h \vdash p = \mathsf{malloc}_\alpha(e) : s[p \mapsto (l_\alpha, 0)] \mid h[(l_\alpha, i) \mapsto w_i]_{0 \leq i < n}}$$

$$\frac{\begin{array}{c} s \mid h \vdash e \Rightarrow v \\ s(p) = (l_\alpha, o) \qquad (l_\alpha, o) \in \mathsf{dom}(h) \end{array}}{s \mid h \vdash *p = e : s \mid h[(l_\alpha, o) \mapsto v]}$$

Fig. 7. Effets de bord sur la mémoire à l'exécution d'une instruction. (Etant donnée f une fonction, $g \stackrel{def}{=} f[x \mapsto v]$ désigne l'application coïncidant avec f sur tout son domaine de définition, sauf en x où $g(x) = v$.)

- l'instruction $p = \mathsf{malloc}_\alpha(e)$ introduit un nouveau bloc l_α dans le tas h. Celui-ci est donc étendu sur les adresses $(l_\alpha, 0), \ldots, (l_\alpha, n-1)$, dans lesquelles sont placées des valeurs aléatoires $w_0, \ldots, w_{n-1}$ (conformément à la section 2.1). De plus, p pointe maintenant vers le début du bloc, d'où $p \mapsto (l_\alpha, 0)$ dans la nouvelle pile. On choisit la location l_α comme distincte de toutes les autres locations déjà présentes dans le tas h : en effet, chaque appel à malloc introduit en effet un bloc de mémoire qui ne chevauche pas ceux déjà alloués. Par ailleurs, on requiert que n soit strictement positif afin que le modèle soit le plus générique possible (par exemple, selon la norme Ansi C, le comportement de $\mathsf{malloc}(0)$ dépend de l'implémentation [23, section 7.20.3, paragraphe 1]).
- enfin, l'écriture dans le tas $*p = e$ place la valeur v de e à l'adresse $s(p) = (l_\alpha, o)$ pointée par p. Il est donc nécessaire que p soit initialisé, et que l'adresse vers laquelle il pointe soit effectivement allouée en mémoire, ce qui correspond à la condition $(l_\alpha, o) \in \mathsf{dom}(h)$.

Sémantique des conditions La sémantique des conditions est définie de manière similaire à celles des instructions : si *cond* est une condition,

$$\frac{\langle i, cond, j \rangle \qquad s \mid h \vdash cond : s' \mid h'}{(i, s \mid h) \rightarrow (j, s' \mid h')} \tag{7}$$

où la relation $s \mid h \vdash cond : s' \mid h'$, définie à la figure 8, a pour effet de sélectionner les états de la mémoire $s \mid h$ vérifiant la condition *cond*. L'état $s' \mid h'$ est alors identique au premier, puisque les conditions n'ont pas d'effet de bord sur la mémoire.

Exemple 4 Considérons le programme $P_{receive}$ dont le graphe de flot de contrôle est donné à la figure 5. Les états $(1, s_1 \mid h_1)$, $(2, s_2 \mid h_2)$, $\ldots$, $(12, s_{12} \mid h_{12})$ définis

$$\frac{s \shortmid h \vdash e \Rightarrow 0}{s \shortmid h \vdash (e == 0) : s \shortmid h} \qquad \frac{s \shortmid h \vdash e \Rightarrow v \quad v \neq 0}{s \shortmid h \vdash !(e == 0) : s \shortmid h}$$

$$\frac{s \shortmid h \vdash e \Rightarrow v \quad v \leq 0}{s \shortmid h \vdash (e \leq 0) : s \shortmid h} \qquad \frac{s \shortmid h \vdash e \Rightarrow v \quad v > 0}{s \shortmid h \vdash !(e \leq 0) : s \shortmid h}$$

Fig. 8. Règles de sélection des états de la mémoire par les conditions.

à la figure 9 constituent une trace possible de l'exécution du programme :

$$(1, s_1 \shortmid h_1) \rightarrow (2, s_2 \shortmid h_2) \rightarrow \ldots \rightarrow (12, s_{12} \shortmid h_{12}) \ .$$

L'état final $(12, s_{12} \shortmid h_{12})$ coïncide avec l'état donné à l'exemple 3, prouvant que ce dernier est bien accessible (l'état $(1, s_1 \shortmid h_1)$ étant bien un état initial possible comme nous le verrons dans la section 3.4). ❑

3.4 Sémantique collectrice d'un programme

Comme son nom l'indique, la sémantique collectrice $\mathcal{C}(P)$ d'un programme P collecte l'ensemble des états atteignables de la machine au cours de l'exécution de P. Ainsi, $\mathcal{C}(P)$ est une partie de States. Pour la définir, nous introduisons la fonction $F : \wp(\mathsf{States}) \rightarrow \wp(\mathsf{States})$ qui correspond à l'exécution d'un pas supplémentaire dans le programme :

$$F(X) \stackrel{def}{=} \{(entry(P), s_0 \shortmid \emptyset) \mid s_0 \in \mathsf{Stack} \wedge \forall p \in \mathsf{PtrVars}.s_0(p) = \omega\} \cup \bigcup_{(i, s \shortmid h) \in X} \{(j, s' \shortmid h') \mid (i, s \shortmid h) \rightarrow (j, s' \shortmid h')\} \ . \quad (8)$$

Intuitivement, $\{(entry(P), s_0 \shortmid \emptyset) \mid s_0 \in \mathsf{Stack} \wedge \forall p \in \mathsf{PtrVars}.s_0(p) = \omega\}$ constitue l'ensemble des états initiaux possibles de la machine : comme spécifié dans la section 2.1, les variables de la pile ne sont pas initialisées[9], et le tas ne contient aucune donnée. Par ailleurs, les ensembles $\{(j, s' \shortmid h') \mid (i, s \shortmid h) \rightarrow (j, s' \shortmid h')\}$ correspondent aux états atteignables à partir des états $(i, s \shortmid h) \in X$ en exécutant une instruction ou en appliquant une condition supplémentaire.

La sémantique collectrice $\mathcal{C}(P)$ est alors définie comme étant la plus petite solution de l'équation $X = F(X)$. On dit que $\mathcal{C}(P)$ est le *plus petit point fixe* de F, ce que nous notons $\mathcal{C}(P) \stackrel{def}{=} \mathsf{lfp}\, F$. L'existence d'un tel point fixe est donnée par le théorème de Tarski [43], et repose sur le fait que $\wp(\mathsf{States})$ forme un treillis complet et que F est une fonction monotone.

[9] Ainsi, les variables de type **uchar** ont des valeurs quelconques dans $\mathbb{N}$, et celles de type pointeur ont la valeur ω.

$$\begin{cases} s_1 : i \mapsto 3, n \mapsto 8, sz \mapsto 53, tmp \mapsto 41, p \mapsto \omega, t \mapsto \omega, \\ h_1 : \end{cases}$$

$$\begin{cases} s_2 : i \mapsto 3, n \mapsto 8, sz \mapsto 53, tmp \mapsto 41, p \mapsto \omega, t \mapsto (\lambda_\alpha, 0), \\ h_2 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_3 : i \mapsto 3, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto \omega, t \mapsto (\lambda_\alpha, 0) \\ h_3 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_5 : i \mapsto 3, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto \omega, t \mapsto (\lambda_\alpha, 0) \\ h_5 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_6 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto \omega, t \mapsto (\lambda_\alpha, 0) \\ h_6 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_7^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto (\lambda_\alpha, 0), t \mapsto (\lambda_\alpha, 0) \\ h_7^0 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_8^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 41, p \mapsto (\lambda_\alpha, 0), t \mapsto (\lambda_\alpha, 0) \\ h_8^0 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_9^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 72, p \mapsto (\lambda_\alpha, 0), t \mapsto (\lambda_\alpha, 0) \\ h_9^0 : (\lambda_\alpha, 0) \mapsto 65, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_{10}^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 72, p \mapsto (\lambda_\alpha, 0), t \mapsto (\lambda_\alpha, 0) \\ h_{10}^0 : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_{11}^0 : i \mapsto 0, n \mapsto 8, sz \mapsto 6, tmp \mapsto 72, p \mapsto (\lambda_\alpha, 1), t \mapsto (\lambda_\alpha, 0) \\ h_{11}^0 : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_7^1 : i \mapsto 1, n \mapsto 8, sz \mapsto 6, tmp \mapsto 72, p \mapsto (\lambda_\alpha, 1), t \mapsto (\lambda_\alpha, 0) \\ h_7^1 : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 23, (\lambda_\alpha, 2) \mapsto 12, (\lambda_\alpha, 3) \mapsto 1, \\ \quad (\lambda_\alpha, 4) \mapsto 234, (\lambda_\alpha, 5) \mapsto 8, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\vdots$$

$$\begin{cases} s_7^6 : i \mapsto 6, n \mapsto 8, sz \mapsto 6, tmp \mapsto 0, p \mapsto (\lambda_\alpha, 6), t \mapsto (\lambda_\alpha, 0) \\ h_7^6 : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 101, (\lambda_\alpha, 2) \mapsto 108, (\lambda_\alpha, 3) \mapsto 108, \\ \quad (\lambda_\alpha, 4) \mapsto 111, (\lambda_\alpha, 5) \mapsto 0, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

$$\begin{cases} s_{12} : i \mapsto 6, n \mapsto 8, sz \mapsto 6, tmp \mapsto 0, p \mapsto (\lambda_\alpha, 6), t \mapsto (\lambda_\alpha, 0) \\ h_{12} : (\lambda_\alpha, 0) \mapsto 72, (\lambda_\alpha, 1) \mapsto 101, (\lambda_\alpha, 2) \mapsto 108, (\lambda_\alpha, 3) \mapsto 108, \\ \quad (\lambda_\alpha, 4) \mapsto 111, (\lambda_\alpha, 5) \mapsto 0, (\lambda_\alpha, 6), \mapsto 179, (\lambda_\alpha, 7) \mapsto 23 \end{cases}$$

Fig. 9. Une trace d'exécution possible du programme P

Comme la fonction F est semi-continue, une version constructive du théorème de Tarski [11] précise que :

$$\mathcal{C}(P) = \bigcup_{n \geq 0} F^n(\emptyset) \ , \tag{9}$$

où F^n désigne l'itérée n-ième de F. Intuitivement, cela signifie que la sémantique concrète est constituée de l'ensemble de tous les états accessibles au bout d'un nombre fini de pas d'exécution.

Une formulation équivalente est de considérer $\mathcal{C}(P)$ comme une application de Ctrl vers $\wp(\mathsf{Mem})$: il suffit alors de collecter à chaque $i \in \mathsf{Ctrl}$ l'ensemble des états de la mémoire possibles au point i. C'est une vision fonctionnelle de la sémantique collectrice. On peut alors redéfinir F comme l'application qui à $X : \mathsf{Ctrl} \to \wp(\mathsf{Mem})$ associe la fonction $F(X) : \mathsf{Ctrl} \to \wp(\mathsf{Mem})$ définie par : pour tout $j \in \mathsf{Ctrl}$,

$$F(X)(j) \stackrel{def}{=} \begin{cases} \{s_0 \natural \emptyset \mid s_0 \in \mathsf{Stack} \wedge \forall p \in \mathsf{PtrVars}.s_0(p) = \omega\} & \text{si } j = \mathit{entry}(P) \\ \displaystyle\bigcup_{s \natural h \in X(i)} \{s' \natural h' \mid (i, s \natural h) \to (j, s' \natural h')\} & \text{sinon} \end{cases} . \tag{10}$$

Et de manière similaire, nous avons toujours $\mathcal{C}(P) = \mathsf{lfp}\, F$, et

$$\mathcal{C}(P) = \bigcup_{n \geq 0} F^n(\emptyset) \ . \tag{11}$$

Dans la suite, nous adopterons, par souci de concision, l'une ou l'autre des formulations.

3.5 Propriétés à vérifier

La sémantique collectrice nous fournissant l'ensemble de tous les états possibles de la machine durant l'exécution d'un programme, nous pouvons maintenant vérifier qu'aucun de ces états ne produit d'erreurs.

La seule erreur à l'exécution possible dans notre formalisme est le dépassement de tampon, dû à l'écriture à une adresse non-allouée dans le tas. Autrement dit, si $(i, s \natural h) \in \mathcal{C}(P)$, et que $\langle i, *p = e, j\rangle$, il y a dépassement de tampon si et seulement si $s(p) = \omega$ ou $s(p) = (l_\alpha, o)$ avec $(l_\alpha, o) \notin \mathsf{dom}(h)$.

Nous pourrions alors introduire un état d'erreur $\Box$ dans l'ensemble States, et ajouter les règles d'inférence :

$$\frac{\langle i, *p = e, j\rangle \qquad s(p) = \omega}{(i, s \natural h) \to \Box} \tag{12}$$

$$\frac{\langle i, *p = e, j\rangle \qquad s(p) = (l_\alpha, o) \notin \mathsf{dom}(h)}{(i, s \natural h) \to \Box} \tag{13}$$

Dès lors, l'absence d'erreur à l'exécution du programme serait garantie par $\Box \notin \mathcal{C}(P)$. Mais un tel choix alourdirait sans réelle utilité le formalisme présenté

dans les parties suivantes. Par conséquent, nous préférons coder l'état d'erreur par l'absence de transition, comme si la machine s'arrêtait dès qu'elle essayait d'écrire dans une zone invalide de la mémoire. L'état d'erreur est alors assimilé à un état inatteignable, et il nous suffira donc de vérifier que pour tout état $(i, s \natural h) \in \mathcal{C}(P)$ tel que $\langle i, *p = e, j\rangle$:

$$\exists (l_\alpha, o) \in \mathsf{Addr}.s(p) = (l_\alpha, o) \in \mathsf{dom}(h) \ , \tag{14}$$

pour montrer que le programme ne provoque pas d'erreurs.

3.6 Vers l'analyse

Nous devons maintenant nous assurer que tous les états de la machine lors de l'exécution du programme à analyser vérifient la propriété (14). Une approche naïve serait la suivante : étant donné P un programme quelconque, « calculer » la sémantique collectrice $\mathcal{C}(P)$ et vérifier la propriété (14) sur chaque état. Naturellement, cette démarche est irréalisable d'après le théorème de Rice [39], dont une des conséquences est l'indécidabilité de la propriété sur notre langage : il n'existe pas d'algorithme, prenant en entrée un programme et déterminant en temps fini si oui ou non la propriété (14) est vérifiée par le programme.

D'ailleurs, de nombreuses difficultés se poseraient pour le calcul de $\mathcal{C}(P)$ pour P quelconque : (i) les parties de $\wp(\mathsf{States})$ ne sont pas représentables en machine, c'est-à-dire qu'elles ne peuvent pas être mises d'une manière générale sous la forme de structures de données stockables en machine,[10] (ii) la fonction de transfert F n'est pas calculable (intuitivement, elle ne peut pas être implémentée sous forme d'algorithme), (iii) et si l'on envisage de faire le calcul par itération (équation (9)), il n'est pas assuré que l'union des itérées de F converge au bout d'un nombre fini d'étapes.

Néanmoins, en réalisant des approximations sur les états de sémantique et les fonctions de transfert, chacun des problèmes ci-dessus peut être résolu. De cette manière, nous obtenons un algorithme qui résout le problème de façon approchée.

Mais ces approximations ne peuvent pas se faire de manière arbitraire : en effet, nous voulons nous assurer de *l'absence* d'erreur à l'exécution. Il est donc nécessaire que ces approximations « n'oublient » aucun état accessible lors d'une exécution. Ces approximations seront donc nécessairement des *sur-approximations*, et de cette manière, l'algorithme approché retournera deux résultats possibles : « il est certain que le programme ne provoque pas d'erreurs » ou « je ne sais pas ».[11] C'est sur ce principe que repose l'interprétation abstraite.

4 Interprétation abstraite

Dans cette partie, nous décrivons la démarche évoquée dans la section 3.6 afin de résoudre le problème posé de manière « approchée ». Un exemple d'abstrac-

[10] Par exemple, elles peuvent ne pas être finies.

[11] Néanmoins, l'algorithme pourra dans certains cas signaler avec certitude une erreur. Nous ne présentons pas ici cette possibilité.

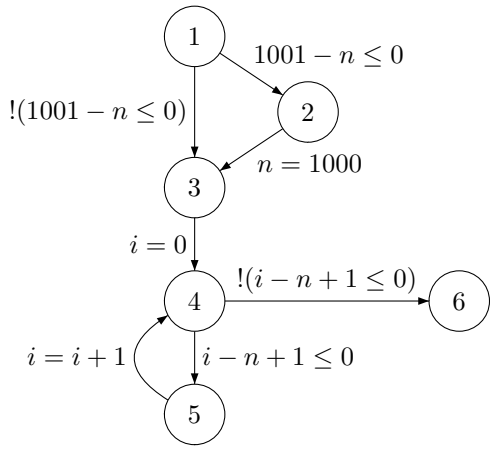
tion est tout d'abord donné comme intuition de cette démarche. Le cadre formel est ensuite défini. Dans un troisième temps, nous introduisons l'abstraction numérique par les polyèdres. Puis nous présentons l'abstraction finale utilisée pour analyser les programmes dans le langage de la partie 2.

4.1 Exemple d'introduction : l'abstraction par des intervalles

Nous décrivons ici le calcul d'une sur-approximation de la sémantique collectrice grâce à des intervalles d'entiers, de manière informelle. Cette technique a été introduite pour la première fois dans [10]. Pour des raisons de simplicité, nous nous restreignons à des programmes ne manipulant que des caractères. De cette manière, l'état de la mémoire est réduit à une pile $s : \mathsf{Vars} \to \mathbb{N}$. Comme illustration du calcul, nous considérons le programme suivant :

```
1 :   if (n > 1000)
2 :     n = 1000;
3 :   i = 0;
4 :   while (i < n) {
5 :     i = i + 1;
6 :   }
```

dont la fonction est de normaliser n à une valeur nécessairement inférieure à 1000, puis d'incrémenter i jusqu'à n. Son graphe de flot de contrôle est le suivant :



Les intervalles d'entiers sont une famille d'ensembles permettant de sur-approximer toute partie X de $\mathbb{Z}$: en effet, si l et u sont respectivement les bornes inférieure et supérieure de X dans $\mathbb{Z} \cup \{-\infty, +\infty\}$, X est bien inclus dans l'intervalle $[l; u]$ (les bornes sont ouvertes selon que l ou u sont égaux à $\pm\infty$). Un tel intervalle peut être d'ailleurs représenté en machine.[12]

[12] En pratique, les entiers à sur-approximer sont au plus codés sur 32 bits (ou 64 bits). Les bornes de ces intervalles peuvent donc être représentées par des données de taille voisine. Il n'y a donc pas à représenter en machine de très grands intervalles, ni même $\pm\infty$.

A partir de cette abstraction élémentaire, nous allons calculer informellement une sur-approximation de la sémantique collectrice. Nous partons pour cela de l'équation (11), et nous calculons une sur-approximation des ensembles $F^n(\emptyset)$, où F est défini comme à l'équation (10). Sous cette convention, les $F^n(\emptyset)$ collectent pour chaque point de contrôle l'ensemble des états de la mémoire possibles au bout de $n-1$ pas d'exécution. Dans notre approximation, plutôt que de collecter un ensemble d'états de la mémoire, nous allons utiliser un état « abstrait » de la mémoire qui va associer à chaque variable du programme un intervalle sur-approximant toutes ses valeurs possibles dans les exécutions réelles. Nous avons ainsi un moyen de représenter en machine une abstraction des $F^n(\emptyset)$.

Exemple 5 Si à un point de contrôle, on a collecté les états suivants :

$$\begin{cases} s_1 : i \mapsto 10, n \mapsto 0 \\ s_2 : i \mapsto 1, \quad n \mapsto 127 \\ s_3 : i \mapsto 7, \quad n \mapsto 132 \end{cases}$$

alors l'ensemble $\{s_1, s_2, s_3\}$ peut être sur-approximé à l'aide d'intervalles, par :

$$\mathcal{S} : i \mapsto [1; 10], n \mapsto [0; 132]$$

❑

Ainsi, $F(\emptyset)$ correspond à l'état initial de la mémoire, avant l'exécution du programme. Comme i et n sont tous deux des caractères non signés, il peut être abstrait en :

$$\mathcal{X}_1 = \begin{cases} 1 : & n \mapsto [0; +\infty[, & i \mapsto [0; +\infty[\\ 2 : & n \mapsto \emptyset, & i \mapsto \emptyset \\ 3 : & n \mapsto \emptyset, & i \mapsto \emptyset \\ 4 : & n \mapsto \emptyset, & i \mapsto \emptyset \\ 5 : & n \mapsto \emptyset, & i \mapsto \emptyset \\ 6 : & n \mapsto \emptyset, & i \mapsto \emptyset \end{cases}$$

où $\emptyset$ désigne un intervalle vide. L'application de la condition $1001 - n \leq 0$ et de sa négation produit intuitivement l'état :

$$\mathcal{X}_2 = \begin{cases} 1 : & n \mapsto [0; +\infty[, & i \mapsto [0; +\infty[\\ 2 : & n \mapsto [1001; +\infty[, & i \mapsto [0; +\infty[\\ 3 : & n \mapsto [0; 1000], & i \mapsto [0; +\infty[\\ 4 : & n \mapsto \emptyset, & i \mapsto \emptyset \\ 5 : & n \mapsto \emptyset, & i \mapsto \emptyset \\ 6 : & n \mapsto \emptyset, & i \mapsto \emptyset \end{cases}$$

car aux points 2 et 3, on ne retrouve que des états de la mémoire s tels que $s(n) > 1000$ et $s(n) \leq 1000$ respectivement. L'étape suivante propage de la

même manière les abstractions :

$$\mathcal{X}_3 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[\\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[\\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[\\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 0] \\ 5: & n \mapsto \emptyset, & i \mapsto \emptyset \\ 6: & n \mapsto \emptyset, & i \mapsto \emptyset \end{cases}.$$

On applique alors les conditions $i + n - 1 \leq 0$ et $!(i + n - 1 \leq 0)$ à l'état, et l'on retrouve donc en 5 les états tels que $n > 0$, et en 6 ceux tels que $n \leq 0$:

$$\mathcal{X}_4 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[\\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[\\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[\\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 0] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 0] \\ 6: & n \mapsto [0; 0]\,, & i \mapsto [0; 0] \end{cases}.$$

Puis l'incrémentation de i entre les points 5 et 4 fait qu'en 4, la nouvelle valeur de i est soit égale à 0 (son ancienne valeur), soit égale à 1. On obtient donc :

$$\mathcal{X}_5 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[\\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[\\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[\\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 0] \\ 6: & n \mapsto [0; 0]\,, & i \mapsto [0; 0] \end{cases},$$

puis :

$$\mathcal{X}_6 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[\\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[\\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[\\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 1] \\ 6: & n \mapsto [0; 1]\,, & i \mapsto [0; 1] \end{cases}.,$$

Et au bout de 1998 itérations supplémentaires, on obtient l'état :

$$\mathcal{X}_{2004} = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[\\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[\\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[\\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 999] \\ 6: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \end{cases}.$$

qui correspond à un point fixe. En effet, une nouvelle itération donnera le même résultat.

Nous avons donc obtenu une sur-approximation du plus petit point fixe de F, c'est-à-dire de $\mathcal{C}(P)$, en un nombre fini d'étapes. Et même si nous n'avons donné qu'une intuition du calcul, les $\mathcal{X}_n$ peuvent être obtenus automatiquement par un algorithme.

Grâce à l'abstraction finale $\mathcal{X}_{2004}$, nous sommes sûrs qu'au point de contrôle 6, la valeur de i sera comprise entre 0 et 1000, quelle que soit la valeur initiale de n. Mais on voit déjà la contrepartie de l'approximation : en effet, nous aurions pu espérer montrer que $i = n$ pour tous les états de la mémoire au point 6.

Par ailleurs, le lecteur peut être surpris du grand nombre d'étapes nécessaires pour obtenir un point fixe. Néanmoins, nous aurions pu être très approximatif pour $\mathcal{X}_5$, et renvoyer plutôt :

$$\mathcal{X}'_5 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[\\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[\\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[\\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 0] \\ 6: & n \mapsto [0; 0]\,, & i \mapsto [0; 0] \end{cases}$$

puis la propagation nous aurait donné :

$$\mathcal{X}'_6 = \begin{cases} 1: & n \mapsto [0; +\infty[\,, & i \mapsto [0; +\infty[\\ 2: & n \mapsto [1001; +\infty[\,, & i \mapsto [0; +\infty[\\ 3: & n \mapsto [0; 1000]\,, & i \mapsto [0; +\infty[\\ 4: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \\ 5: & n \mapsto [1; 1000]\,, & i \mapsto [0; 999] \\ 6: & n \mapsto [0; 1000]\,, & i \mapsto [0; 1000] \end{cases}.$$

Ainsi, au bout de 6 étapes, nous aurions atteint le même point fixe qu'en étant plus précis pendant 2004 étapes. Ce type de technique est appelé *élargissement*, et permet non seulement d'accélérer la convergence du calcul, mais aussi de la garantir en un nombre fini d'étapes. Nous formalisons cette notion ainsi que le principe plus général de l'abstraction dans la partie suivante.

4.2 Formalisation de l'abstraction

Nous présentons ici un modèle mathématique possible pour l'interprétation abstraite. D'autres formalismes envisageables sont traités dans [12].

Principe de base Soit D un ensemble dont nous voulons abstraire les éléments. Nous faisons l'hypothèse que D est muni d'une structure de treillis complet $(D, \preceq, \bot, \top, \curlyvee, \curlywedge)$: $\preceq$ est un ordre partiel sur les éléments de D, $\bot$ et $\top$ sont respectivement les plus petit et plus grand éléments de D, et $\curlyvee$ et $\curlywedge$ sont des opérateurs d'union et d'intersection respectivement.[13] L'ensemble D sera par la suite appelé *domaine concret*.

[13] On a, en particulier, pour tout $(X, Y) \in D^2$: $X \curlywedge Y \preceq X \preceq X \curlyvee Y$ et $X \curlywedge Y \preceq Y \preceq X \curlyvee Y$.

L'abstraction consiste en la donnée d'un *domaine abstrait* $\mathcal{D}$, lui aussi muni d'un ordre partiel $\sqsubseteq$, et d'un *opérateur de concrétisation* $\gamma : \mathcal{D} \to D$ monotone, c'est-à-dire :

$$\forall \mathcal{X}, \mathcal{Y} \in \mathcal{D}. \text{ si } \mathcal{X} \sqsubseteq \mathcal{Y}, \text{alors } \gamma(\mathcal{X}) \preceq \gamma(\mathcal{Y}) \tag{15}$$

Intuitivement, l'ensemble $\mathcal{D}$ est constitué des éléments abstraits, et à chaque élément $\mathcal{X} \in \mathcal{D}$ correspond un élément concret $\gamma(\mathcal{X})$. L'ordre $\sqsubseteq$ peut être compris en terme de précision : ainsi, si $\mathcal{X} \sqsubseteq \mathcal{Y}$, alors $\mathcal{X}$ est plus précis que $\mathcal{Y}$, car il représente un élément concret « plus petit » pour l'ordre $\preceq$.

Exemple 6 Dans l'exemple donné dans la section 4.1, le domaine concret D est l'ensemble $\wp(\mathsf{States})$, muni de l'inclusion $\subseteq$, et le domaine abstrait est défini par :

$$\mathcal{D} = \left(\mathcal{R}(\mathbb{Z})^{\mathsf{Vars}}\right)^{\mathsf{Ctrl}}$$

où $\mathcal{R}(\mathbb{Z})$ désigne l'ensemble des intervalles sur $\mathbb{Z}$, c'est-à-dire :

$$\begin{aligned}\mathcal{R}(\mathbb{Z}) = \{\emptyset\} \cup \{[l;u] \mid l,u \in \mathbb{Z} \wedge l < u\} \\ \cup \{]-\infty;u] \mid u \in \mathbb{Z}\} \cup \{[l;+\infty[\mid l \in \mathbb{Z}\} \cup \{]-\infty;+\infty[\} \ .\end{aligned}$$

Autrement dit, un élément $\mathcal{X} \in \mathcal{D}$ associe à chaque point de contrôle i, un état abstrait de la mémoire $\mathcal{X}(i)$, à savoir une application de Vars dans $\mathcal{R}(\mathbb{Z})$. Avec cette définition, on peut introduire l'ordre $\sqsubseteq$ par :

$$\mathcal{X} \sqsubseteq \mathcal{Y} \text{ si } \forall i \in \mathsf{Ctrl}. \forall x \in \mathsf{Vars}. \mathcal{X}(i)(x) \subseteq \mathcal{Y}(i)(x) \ .$$

Et la concrétisation est donnée par :

$$\begin{aligned}\gamma(\mathcal{X}) : \mathsf{Ctrl} &\to \wp(\mathsf{Mem}) \\ i &\mapsto \{s \mid \forall x \in \mathsf{Vars}. s(x) \in \mathcal{X}(i)(x)\} \ .\end{aligned}$$

❑

Opérateurs abstraits corrects D'une manière générale, si F est une application de D dans lui-même, il est intéressant de construire un « équivalent » abstrait de F sur $\mathcal{D}$. Mais cet équivalent doit bien correspondre à une « sur-approximation ». Ainsi, si $\mathcal{F} : \mathcal{D} \to \mathcal{D}$, nous dirons que $\mathcal{F}$ est *correct* par rapport à F si la condition suivante est vérifiée :

$$\forall \mathcal{X} \in \mathcal{D}. F(\gamma(\mathcal{X})) \preceq \gamma(\mathcal{F}(\mathcal{X})) \ . \tag{16}$$

Autrement dit, $\mathcal{F}(\mathcal{X})$ est bien une sur-approximation de $F(\gamma(\mathcal{X}))$. L'application de $\mathcal{F}$ sur $\mathcal{X}$ n'a donc pas « oublié » d'éléments concrets.

On peut ensuite généraliser la construction d'opérateurs abstraits corrects pour les opérations ou éléments usuels de D. Par exemple,

- pour l'union $\curlyvee$, nous supposerons l'existence d'un opérateur $\sqcup : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ tel que :

$$\gamma(\mathcal{X}) \curlyvee \gamma(\mathcal{Y}) \preceq \gamma(\mathcal{X} \sqcup \mathcal{Y}) \ . \tag{17}$$

– de même, si pour l'intersection $\curlywedge$, on supposera $\mathcal{D}$ muni d'un opérateur $\sqcap$ tel que :
$$\gamma(\mathcal{X}) \curlywedge \gamma(\mathcal{Y}) \preceq \gamma(\mathcal{X} \sqcap \mathcal{Y}) \ . \tag{18}$$
– et pour $\pm$ et $\mp$, nous ferons l'hypothèse que $\mathcal{D}$ a un plus petit élément $\bot$ et un plus grand élément $\top$ pour l'ordre $\sqsubseteq$.

Exemple 7 Le domaine $D = \wp(\mathsf{States})$ étant un treillis complet, il dispose naturellement des opérations d'union $\cup$ et d'intersection $\cap$, ainsi que d'un plus petit élément $\emptyset$ et d'un plus grand élément States. On peut définir des opérateurs abstraits corrects de la façon suivante : pour tout $i \in \mathsf{Ctrl}$,

$$\begin{aligned}
(\mathcal{X} \sqcup \mathcal{Y})(i) &\stackrel{def}{=} x \mapsto \mathcal{X}(i)(x) \bowtie \mathcal{Y}(i)(x) \ , \\
(\mathcal{X} \sqcap \mathcal{Y})(i) &\stackrel{def}{=} x \mapsto \mathcal{X}(i)(x) \cap \mathcal{Y}(i)(x) \ , \\
\bot(i) &\stackrel{def}{=} \lambda i.x \mapsto \emptyset \ , \\
\top(i) &\stackrel{def}{=} \lambda i.x \mapsto \,]-\infty;+\infty[
\end{aligned}$$

où $\mathcal{X}(i) \bowtie \mathcal{Y}(i)$ désigne le plus petit intervalle contenant à la fois $\mathcal{X}(i)$ et $\mathcal{Y}(i)$, et où $x \mapsto f(x)$ représente la fonction qui à tout x associe $f(x)$. ❑

Calcul de points fixes abstraits Supposons maintenant que F est une fonction monotone sur le treillis complet $(D, \preceq, \pm, \mp, \curlyvee, \curlywedge)$, et que pour toute suite $X_0, \ldots, X_n, \ldots$ croissante d'éléments de D, on ait $F\left(\bigvee_n X_n\right) = \bigvee_n F(X_n)$ (F est dite *semi-continue*). Le théorème de Tarski [43] assure alors l'existence du plus petit point fixe pour F, et par une version constructive du théorème [11], nous avons :
$$\mathsf{lfp}\, F = \bigvee_{n \geq 0} F^n(\pm) \ . \tag{19}$$

Si $\mathcal{F}$ est une abstraction correcte et monotone de F, on peut alors surapproximer chaque itéré $F^n(\pm)$ par $\mathcal{F}^n(\bot)$. En effet, nous avons par définition $\pm \preceq \gamma(\bot)$, et par récurrence, si pour un n donné, $F^n(\pm) \preceq \gamma(\mathcal{F}^n(\bot))$, alors

$$\begin{aligned}
F^{n+1}(\pm) &= F(F^n(\pm)) \\
&\preceq F(\gamma(\mathcal{F}^n(\bot))) && \text{par hypothèse de récurrence et } F \text{ monotone} \\
&\preceq \gamma(\mathcal{F}(\mathcal{F}^n(\bot))) && \text{par l'équation (16)} \\
&\preceq \gamma(\mathcal{F}^{n+1}(\bot)) \ .
\end{aligned}$$

Par ailleurs, la suite des $\mathcal{F}^n(\bot)$ est clairement croissante. Ainsi, si cette suite est stationnaire, c'est-à-dire qu'il existe un rang N à partir duquel tous ses termes sont égaux (nous dirons aussi que la suite converge en un nombre fini N d'étapes), on a clairement :
$$\mathsf{lfp}\, F \preceq \gamma(\mathcal{F}^N(\bot)) \tag{20}$$

Nous étions dans cette situation pour le calcul réalisé en 4.1. Mais dans le cas général, rien n'assure que cette suite soit stationnaire. Nous introduisons donc un nouvel opérateur $\nabla : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$, dit *d'élargissement*, qui possède la propriété suivante :

- $\forall \mathcal{X}, \mathcal{Y} \in \mathcal{D}. \mathcal{X} \sqcup \mathcal{Y} \sqsubseteq \mathcal{X} \nabla \mathcal{Y}$.
- pour toute suite croissante d'éléments $\mathcal{X}_0 \preceq \ldots \preceq \mathcal{X}_n \preceq \ldots$, la suite définie par

$$\begin{cases} \mathcal{Y}_0 \stackrel{def}{=} \mathcal{X}_0 \\ \mathcal{Y}_{n+1} \stackrel{def}{=} \mathcal{Y}_n \nabla \mathcal{X}_{n+1} \end{cases} \tag{21}$$

  converge en un nombre fini d'étapes.

Ainsi, l'opérateur ∇ va nous permettre d'assurer la convergence en un nombre fini d'étapes.

Théorème 1 *Si la suite* $(\mathcal{X}_n)_n$ *est définie par :*

$$\begin{cases} \mathcal{X}_0 \stackrel{def}{=} \bot \\ \mathcal{X}_{n+1} \stackrel{def}{=} \begin{cases} \mathcal{X}_n & \text{si } \mathcal{F}(\mathcal{X}_n) \sqsubseteq \mathcal{X}_n \\ \mathcal{X}_n \nabla \mathcal{F}(\mathcal{X}_n) & \text{sinon} \end{cases} \end{cases}, \tag{22}$$

alors cette suite est stationnaire, et sa limite $\mathcal{X}_N$ *vérifie :*

$$\mathsf{lfp}\, F \preceq \gamma(\mathcal{X}_N)\ . \tag{23}$$

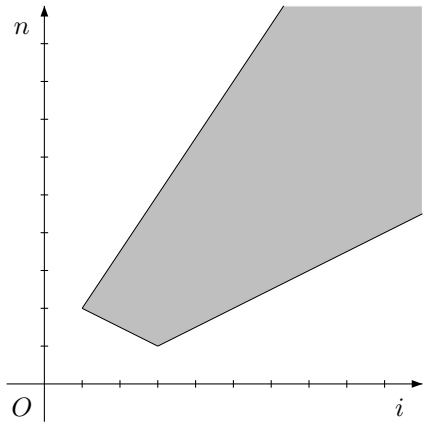
Par conséquent, si les éléments du domaine abstrait $\mathcal{D}$ sont représentables en machine, si la fonction $\mathcal{F}$ est calculable, et si $\mathcal{D}$ est muni d'un opérateur d'élargissement ∇ lui aussi calculable, le théorème 1 nous donne un algorithme pour calculer une sur-approximation de $\mathsf{lfp}\, F$ *en temps fini*.

4.3 Les polyèdres convexes

Nous présentons dans cette partie un domaine abstrait numérique qui permet d'approximer un ensemble de valeurs numériques par des relations d'inégalités affines entre elles, c'est-à-dire par des polyèdres convexes. Une telle abstraction nous permettra de découvrir des invariants plus précis que les intervalles. Elle a été pour la première fois définie dans [13].

Considérons un ensemble $V = \{V_1, \ldots, V_d\}$ fini et ordonné de variables deux à deux distinctes. Nous noterons $\mathbb{CP}(V)$ l'ensemble des polyèdres convexes dans l'espace affine $\mathbb{R}^d$, dont les dimensions sont étiquetées par les variables V_1, ..., V_d respectivement. Un polyèdre $\mathcal{P} \in \mathbb{CP}(V)$ peut être représenté sous forme de contraintes d'inégalités affines. C'est pourquoi nous désignons parfois un polyèdre directement par un ensemble de contraintes.

Exemple 8 Pour $d = 2$, considérons le polyèdre ouvert sur les variables i et n, où i est en abscisse et n en ordonnée :



Il est défini par le système de contrainte suivante :

$$\begin{cases} 2n - 3i \leq 1 \\ 2n + i \geq 5 \\ 2n - i \geq -1 \end{cases}$$

❑

L'ensemble $\mathbb{CP}(V)$ muni de l'inclusion $\subseteq$ forme un ordre partiel, et on peut introduire un opérateur de concrétisation $\gamma_{poly} : \mathbb{CP}(V) \to \mathbb{R}^V$ défini par :

$$\gamma_{poly}(\mathcal{P}) = \{f : V \to \mathbb{R} \mid (f(V_1), \ldots, f(V_d)) \in \mathcal{P}\} \tag{24}$$

Naturellement, γ_{poly} est bien monotone. Les opérateurs usuels peuvent être définis de la manière suivante :

- l'union de deux polyèdres $\mathcal{P}_1$ et $\mathcal{P}_2$ peut être sur-approximée par l'enveloppe convexe $\mathcal{P}_1 \uplus \mathcal{P}_2$ de $\mathcal{P}_1 \cup \mathcal{P}_2$ (car $\mathcal{P}_1 \cup \mathcal{P}_2$ n'est pas nécessairement un polyèdre).
- l'intersection de $\mathcal{P}_1$ et $\mathcal{P}_2$ peut être abstraite en $\mathcal{P}_1 \cap \mathcal{P}_2$ (les polyèdres sont stables par intersection).
- les ensembles $\emptyset$ et $\mathbb{R}^d$ sont respectivement les plus petit et plus grand éléments.

Nous allons maintenant définir quelques transformations abstraites sur les polyèdres. Nous nous restreindrons à une présentation intuitive par souci de simplicité.

- L'application sur un polyèdre $\mathcal{P}$ d'une (in)égalité $c_0 + \sum_i c_i V_i \diamond 0$, où $\diamond \in \{\leq, =, \geq\}$, consiste à intersecter $\mathcal{P}$ avec le polyèdre défini par l'(in)égalité. Si $(\!|c_0 + \sum_i c_i V_i \diamond 0|\!)(\mathcal{P})$ désigne le polyèdre résultant, on a bien :

$$\left\{ f : V \to \mathbb{R} \mid (f(V_1), \ldots, f(V_d)) \in \mathcal{P} \wedge \left(c_0 + \sum_i c_i f(V_i) \diamond 0 \right) \right\}$$
$$\subseteq \gamma_{poly} \left((\!|c_0 + \sum_i c_i V_i \diamond 0|\!)(\mathcal{P}) \right) . \tag{25}$$

– L'affectation d'une valeur aléatoire à une variable $V_i \leftarrow ?$ s'obtient en prenant l'enveloppe convexe de l'union du polyèdre $\mathcal{P}$ avec la droite passant par l'origine et dirigée selon la i-ème coordonnée. On a alors :

$$\{f : V \to \mathbb{R} \mid \exists v.(f(V_1), \ldots, f(V_{i-1}), v, f(V_{i+1}), \ldots, f(V_d)) \in \mathcal{P}\} \subseteq \gamma_{poly}\left(\llparenthesis V_i \leftarrow ? \rrparenthesis(\mathcal{P})\right) \ . \quad (26)$$

– L'affectation $V_i \leftarrow c_0 + \sum_j c_j V_j$ d'une expression affine à une variable V_i consiste en :
  – lorsque $c_i \neq 0$, remplacer la variable V_i par l'expression $\frac{V_i - c_0 - \sum_{j \neq i} c_j V_j}{c_i}$ dans le système de contraintes définissant $\mathcal{P}$.
  – lorsque $c_i = 0$, affecter tout d'abord une valeur aléatoire à V_i, de manière à « annuler » l'ancienne valeur de V_i, puis appliquer au polyèdre résultant la condition $V_i - \left(c_0 + \sum_j c_j V_j\right) = 0$.

  Dans les deux cas, on obtient un polyèdre $\llparenthesis V_i \leftarrow c_0 + \sum_j c_j V_j \rrparenthesis(\mathcal{P})$ qui vérifie :

$$\left\{ g : V \to \mathbb{R} \mid \begin{array}{c} (f(V_1), \ldots, f(V_d)) \in \mathcal{P} \wedge \forall j \neq i.g(V_j) = f(V_i) \\ \wedge\ g(V_i) = c_0 + \sum_j c_j f(V_j) \end{array} \right\} \subseteq \gamma_{poly}\left(\llparenthesis V_i \leftarrow c_0 + \sum_j c_j V_j \rrparenthesis(\mathcal{P})\right) \ . \quad (27)$$

Ces fonctions de transfert vont nous être utiles pour construire une abstraction correcte de la fonction F introduite à la section 3.4 et ce, pour n'importe quel programme.

Enfin, nous pouvons introduire opérateur d'élargissement ∇_{poly}, où, si $\mathcal{P}, \mathcal{Q} \in \mathbb{CP}(V)$, $\mathcal{P} \nabla \mathcal{Q}$ est défini par le système de contraintes suivant :

$$\{C \mid C \text{ contrainte de } \mathcal{P} \text{ entièrement vérifiée dans } \mathcal{Q}\} \cup \{C' \mid C' \text{ contrainte de } \mathcal{Q} \text{ équivalente à une contrainte de } \mathcal{P}\} \ . \quad (28)$$

Une contrainte C' est équivalente à une contrainte de $\mathcal{P}$ s'il existe une contrainte C'' de $\mathcal{P}$ telle que $\mathcal{P}$ reste inchangé en remplaçant C'' par C' :

$$\mathcal{P} = (\mathcal{P} \setminus \{C''\}) \cup \{C'\} \ .$$

Nous renvoyons le lecteur à [13] pour des explications supplémentaires sur ∇_{poly}.

Naturellement, les éléments de $\mathbb{CP}(V)$ sont représentables en machine, et les opérations « géométriques » utilisées dans la définition des opérations abstraites sont toutes calculables. Muni de l'opérateur ∇_{poly}, cette abstraction est donc adaptée au calcul de sur-approximations. Le domaine des polyèdres est plus précis que celui des intervalles, par exemple il nous aurait permis de découvrir l'invariant $i = n$ au point de contrôle 6 dans le programme présenté en 4.1. Néanmoins, cette précision a un prix, car les opérations peuvent se révéler très coûteuses : en temps comme en mémoire, elles ont une complexité exponentielle

$$\mathsf{CharVars} \stackrel{def}{=} \{x \in \mathsf{Vars} \mid x \text{ de type } \mathbf{uchar}\} \quad \mathsf{PtrVars} \stackrel{def}{=} \{p \in \mathsf{Vars} \mid z \text{ de type } \mathbf{char}*\}$$
$$\mathsf{Alloc} \stackrel{def}{=} \{\alpha \mid p = \mathsf{malloc}_\alpha(e)\}$$

$$\mathsf{LocVars} \stackrel{def}{=} \{p_\ell \mid p \in \mathsf{PtrVars}\} \qquad \mathsf{OffVars} \stackrel{def}{=} \{p_o \mid p \in \mathsf{PtrVars}\}$$
$$\mathsf{SizeVars} \stackrel{def}{=} \{\alpha_s \mid \alpha \in \mathsf{Alloc}\} \qquad \mathsf{NumVars} \stackrel{def}{=} \mathsf{CharVars} \cup \mathsf{OffVars} \cup \mathsf{SizeVars}$$

Fig. 10. Les variables symboliques et numériques.

en le nombre de variables d. Il existe cependant d'autres domaines décrits dans la littérature [32,34,35,42] qui permettraient d'obtenir un meilleur compromis précision/complexité pour l'analyse.

4.4 Abstraction finale

Nous avons maintenant tous les éléments nécessaires pour la construction d'une abstraction de $\wp(\mathsf{States})$, en vue d'une analyse de programmes écrits dans le langage présenté en partie 2.

Rappelons que $\mathsf{CharVars}$ et $\mathsf{PtrVars}$ désignent respectivement l'ensemble des variables de type **uchar** et pointeur, et Alloc l'ensemble des α étiquetant les instructions $p = \mathsf{malloc}_\alpha(e)$. Pour chaque variable p de type pointeur, nous construisons deux variables, p_ℓ et p_o, représentant intuitivement la location et la composante de décalage de l'adresse pointée par p respectivement. Les variables p_ℓ forment l'ensemble $\mathsf{LocVars}$, les variables p_o l'ensemble $\mathsf{OffVars}$. Pour chaque symbole α étiquetant une instruction $p = \mathsf{malloc}_\alpha(e)$, nous introduisons une variable α_s correspondant à la taille du bloc mémoire alloué en α. Les α_s constituent quant à elles l'ensemble $\mathsf{SizeVars}$. Enfin, les variables de $\mathsf{CharVars}$, $\mathsf{OffVars}$ et $\mathsf{SizeVars}$ étant toutes de nature numérique, nous les regroupons dans un ensemble noté $\mathsf{NumVars}$. La figure 10 récapitule ces notations.

Nous prendrons pour domaine abstrait $\mathsf{AStates}$ l'ensemble des applications de Ctrl dans AMem, où AMem désigne l'ensemble des états abstraits de la mémoire, et dont les éléments sont de la forme $(\mathcal{L}, \mathcal{N})$, où $\mathcal{L} : \mathsf{LocVars} \to \wp(\mathsf{Alloc} \cup \{\omega\})$ et $\mathcal{N} \in \mathbb{CP}(\mathsf{NumVars})$. Autrement dit :

$$\mathsf{AStates} \stackrel{def}{=} \mathsf{AMem}^{\mathsf{Ctrl}} \qquad \mathsf{AMem} \stackrel{def}{=} \left(\wp(\mathsf{Alloc} \cup \{\omega\})^{\mathsf{LocVars}}\right) \times \mathbb{CP}(\mathsf{NumVars}) \ . \tag{29}$$

Ainsi, un état abstrait associe à chaque point de contrôle i une représentation abstraite de la mémoire $(\mathcal{L}_i, \mathcal{N}_i)$. La fonction $\mathcal{L}_i$ associe à chaque pointeur une location abstraite, ici sous la forme d'une partie de $\mathsf{Alloc} \cup \{\omega\}$. Le polyèdre $\mathcal{N}_i$ consiste quant à lui en une sur-approximation des données numériques de la mémoire. Les éléments de $\mathsf{AStates}$ sont bien représentables en machine car Ctrl, Alloc, $\mathsf{LocVars}$ et $\mathsf{NumVars}$ sont tous des ensembles finis.

L'ordre partiel $\sqsubseteq$ sur $\mathsf{AStates}$ dérive de l'inclusion $\subseteq$ sur les domaines $\wp(\mathsf{Alloc} \cup \{\omega\})$ et $\mathbb{CP}(\mathsf{NumVars})$, c'est-à-dire : $\mathcal{X} \sqsubseteq \mathcal{Y}$ si pour tout $i \in \mathsf{Ctrl}$,

$$\begin{cases} \forall p_\ell \in \mathsf{LocVars}. \mathcal{L}_i(p_\ell) \subseteq \mathcal{L}'_i(p_\ell) \\ \mathcal{N}_i \subseteq \mathcal{N}'_i \end{cases}, \tag{30}$$

où $\mathcal{X}(i) = (\mathcal{L}_i, \mathcal{N}_i)$ et $\mathcal{Y}(i) = (\mathcal{L}'_i, \mathcal{N}'_i)$.

L'opérateur de concrétisation $\gamma : \mathsf{AStates} \to \wp(\mathsf{States})$ est défini de la manière suivante :

$$\gamma(\mathcal{X}) \stackrel{def}{=} \begin{cases} \mathsf{Ctrl} \to \wp(\mathsf{Mem}) \\ \quad i \mapsto \gamma_{mem}(\mathcal{X}(i)) \end{cases}, \tag{31}$$

où $\gamma_{mem} : \mathsf{AMem} \to \wp(\mathsf{Mem})$ exprime le sens d'un état abstrait de la mémoire $(\mathcal{L}, \mathcal{N})$:

$$\gamma_{mem}(\mathcal{L}, \mathcal{N}) \stackrel{def}{=}$$
$$\left\{ \begin{array}{l} \qquad s \natural h \in \mathsf{States} \wedge \nu \in \gamma_{poly}(\mathcal{N}) \\ \qquad \wedge \forall x \in \mathsf{CharVars}. s(x) = \nu(x) \\ s \natural h \mid \wedge \forall p \in \mathsf{PtrVars}. \left[\begin{array}{l} (\alpha \in \mathcal{L}(p_\ell) \cap \mathsf{Alloc} \wedge s(p) = (l_\alpha, \nu(p_o))) \\ \vee\ (\omega \in \mathcal{L}(p_\ell) \wedge s(p) = \omega) \end{array} \right] \\ \qquad \wedge \forall \alpha \in \mathsf{Alloc}. \forall l_\alpha \in \mathsf{dom}(h). \text{il existe } k_{l_\alpha} \text{ vérifiant} \\ \qquad\quad ((l_\alpha, o) \in \mathsf{dom}(h) \Leftrightarrow 0 \leq o < k_{l_\alpha}) \wedge \nu[\alpha_s \mapsto k_{l_\alpha}] \in \gamma_{poly}(\mathcal{N}) \end{array} \right\}. \tag{32}$$

Intuitivement, pour chaque point de contrôle $i \in \mathsf{Ctrl}$, si $\mathcal{X}(i) = (\mathcal{L}_i, \mathcal{N}_i)$,

- $\mathcal{L}_i$ sur-approxime les locations des blocs pointés par les pointeurs du programme : $s(p)$ peut être de la forme $(l_\alpha, \cdot)$ avec $\alpha \in \mathcal{L}_i(p_\ell) \cap \mathsf{Alloc}$, ou peut ne pas être initialisé si $\omega \in \mathcal{L}_i(p_\ell)$.
- $\mathcal{N}_i$ sur-approxime toutes les données numériques : les valeurs des caractères $(s(x) = \nu(x))$, les décalages des adresses pointées par les pointeurs $(s(p) = (\cdot, \nu(p_o)))$, et les tailles des blocs dans le tas : si le bloc de location l_α est présent dans le tas, sa taille k_{l_α} est sur-approximée par α_s $(\nu[\alpha_s \mapsto k_{l_\alpha}] \in \gamma_{poly}(\mathcal{N}))$.

Le principe de l'approximation sur le tas est donc d'oublier totalement le contenu exact du tas, pour ne retenir que la taille des blocs alloués, et de fusionner l'information sur les blocs alloués à un même site d'allocation α.

Exemple 9 Considérons l'état abstrait de la mémoire suivant :

$$(\mathcal{L}, \mathcal{N}) = \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq p_o = i = sz \leq \alpha_s = n\} \end{array} \right)$$

Il représente les états $s \natural h$ de la mémoire tels que :

- $s(n) \geq 1$, $0 \leq s(i) \leq n$ et $s(tmp) \geq 0$.
- $s(p) = (\lambda_\alpha, s(i))$ et $s(t) = (\lambda'_\alpha, 0)$, avec $\lambda_\alpha, \lambda'_\alpha \in \mathsf{Loc}$ (λ_α et λ'_α peuvent être égaux).

– le tas h est constitué de deux blocs λ_α et λ'_α qui peuvent être confondus. Chacun de ces blocs a une taille égale à $s(n)$. En effet, la taille k_{λ_α} du bloc λ_α, définie par $(\lambda_\alpha, o) \in \mathsf{dom}(h) \Leftrightarrow 0 \leq o < k_{\lambda_\alpha}$, vérifie l'égalité $k_{\lambda_\alpha} = s(n)$. De même pour le bloc λ'_α.

Des opérateurs abstraits usuels sur $\mathsf{AStates}$ peuvent être définis à partir de ceux sur $\wp(\mathsf{Alloc} \cup \{\omega\})$ et $\mathbb{CP}(\mathsf{NumVars})$:

– un opérateur abstrait d'union $\sqcup$, tel que pour tout $i \in \mathsf{Ctrl}$, si $\mathcal{X}(i) = (\mathcal{L}_i, \mathcal{N}_i)$ et $\mathcal{Y}(i) = (\mathcal{L}'_i, \mathcal{N}'_i)$, alors

$$(\mathcal{X} \sqcup \mathcal{Y})(i) \stackrel{def}{=} (\mathcal{L}_i, \mathcal{N}_i) \dot{\sqcup} (\mathcal{L}'_i, \mathcal{N}'_i) \ , \tag{33}$$

où $(\mathcal{L}, \mathcal{N}) \dot{\sqcup} (\mathcal{L}', \mathcal{N}')$ est lui-même un opérateur abstrait d'union sur AMem :

$$(\mathcal{L}, \mathcal{N}) \dot{\sqcup} (\mathcal{L}', \mathcal{N}') \stackrel{def}{=} (p_\ell \mapsto \mathcal{L}(p_\ell) \cup \mathcal{L}'(p_\ell), \mathcal{N} \uplus \mathcal{N}') \ . \tag{34}$$

– le plus petit élément $\bot$, défini par $\forall i \in \mathsf{Ctrl}. \bot(i) \stackrel{def}{=} (p_\ell \mapsto \emptyset, \emptyset)$.
– un opérateur d'élargissement ∇, où pour tout $i \in \mathsf{Ctrl}$, si $\mathcal{X}(i) = (\mathcal{L}_i, \mathcal{N}_i)$ et $\mathcal{Y}(i) = (\mathcal{L}'_i, \mathcal{N}'_i)$,

$$(\mathcal{X} \nabla \mathcal{Y})(i) \stackrel{def}{=} (p_\ell \mapsto \mathcal{L}_i(p_\ell) \cup \mathcal{L}'_i(p_\ell), \mathcal{N}_i \nabla_{poly} \mathcal{N}'_i)$$

Leur correction résulte de celle des opérations sous-jacentes sur $\wp(\mathsf{Alloc} \cup \{\omega\})$ et $\mathbb{CP}(\mathsf{NumVars})$. Par ailleurs, pour l'opérateur d'élargissement ∇, l'union classique $\cup$ sur la composante des locations est suffisante pour assurer la convergence en un nombre fini d'itérations, car $\mathsf{Alloc} \cup \{\omega\}$ est un ensemble fini, et donc la hauteur de $\wp(\mathsf{Alloc} \cup \{\omega\})$ est finie. D'autres opérateurs (intersection, plus grand élément) pourraient être construits, mais ils ne seront pas nécessaires dans la suite.

Etant donné un programme P quelconque, nous allons maintenant construire une abstraction correcte $\mathcal{F}$ de la fonction de transfert F définie à l'équation (10). Nous définissons pour cela : étant donné $\mathcal{X} \in \mathsf{AStates}$, pour tout $j \in \mathsf{Ctrl}$,

$$\mathcal{F}(\mathcal{X})(j) \stackrel{def}{=} \begin{cases} \left(p_\ell \mapsto \{\omega\}, \bigcap_{x, \alpha_s} \{x \geq 0\} \cap \{\alpha_s = 0\} \right) & \text{si } j = \mathit{entry}(P) \\ \dot{\bigsqcup}_{\langle i, \mathit{stmt}, j \rangle} [\![\mathit{stmt}]\!](\mathcal{L}_i, \mathcal{N}_i) & \text{sinon} \end{cases} , \tag{35}$$

où $[\![\mathit{stmt}]\!] : \mathsf{AMem} \to \mathsf{AMem}$ est une fonction que nous allons définir ci-après, et qui vérifie, pour toute instruction ou condition *stmt* :

$$\{s' \natural h' \mid s \natural h \in \gamma_{mem}(\mathcal{L}, \mathcal{N}) \wedge s \natural h \vdash \mathit{stmt} : s' \natural h'\} \subseteq \gamma_{mem}\left([\![\mathit{stmt}]\!](\mathcal{L}, \mathcal{N})\right) \ . \tag{36}$$

Autrement dit, $[\![\mathit{stmt}]\!]$ doit être définie de façon à être une abstraction correcte de l'effet de l'instruction ou de la condition *stmt* dans la sémantique concrète.

De façon évidente, l'état initial abstrait de la mémoire sur-approxime l'état concret correspondant :

$$\{s_0 \natural \emptyset \mid s_0 \in \mathsf{Stack} \wedge \forall p \in \mathsf{PtrVars}.s(p) = \omega\} \subseteq \gamma_{mem}\left(p_\ell \mapsto \{\omega\}, \bigcap_{x,\alpha_s} \{x \geq 0\} \cap \{\alpha_s = 0\}\right) , \quad (37)$$

aussi, si la relation 36 est vérifiée, on en déduit la propriété suivante :

Proposition 1. *La fonction de transfert abstraite $\mathcal{F}$ est une abstraction correcte de la fonction de transfert concrète F.*

Il nous reste à définir précisément $[\![stmt]\!]$, notamment grâce aux opérateurs abstraits sur les polyèdres présentés en 4.3.

$$[\![x = e]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}, (\!|x \leftarrow e|\!)(\mathcal{N})) \quad (38)$$

$$[\![x = \mathsf{getuchar}()]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}, (\!|x \geq 0|\!) \circ (\!|x \leftarrow ?|\!)(\mathcal{N})) \quad (39)$$

$$[\![p = q + e]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}[p_\ell \mapsto \mathcal{L}(q_\ell)], (\!|p_o \leftarrow q_o + e|\!)(\mathcal{N})) \quad (40)$$

$$[\![p = \mathsf{malloc}_\alpha(e)]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}[p_\ell \mapsto \{\alpha\}], (\!|p_o \leftarrow 0|\!)(\mathcal{N}')) \text{ où } \mathcal{N}' = (\!|\alpha_s \leftarrow e|\!) \circ (\!|e \geq 1|\!)(\mathcal{N}) \uplus (\!|\alpha_s \geq 1|\!)(\mathcal{N}) \quad (41)$$

$$[\![*p = e]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}[p_\ell \mapsto \mathcal{L}(p_\ell) \cap \mathsf{Alloc}], \mathcal{N}) \quad (42)$$

$$[\![e \leq 0]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}, (\!|e \leq 0|\!)(\mathcal{N})) \quad (43)$$

$$[\![e == 0]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}, (\!|e = 0|\!)(\mathcal{N})) \quad (44)$$

$$[\![!(e \leq 0)]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}, (\!|e \geq 1|\!)(\mathcal{N})) \quad (45)$$

$$[\![!(e == 0)]\!](\mathcal{L},\mathcal{N}) \stackrel{def}{=} (\mathcal{L}, (\!|e \leq -1|\!)(\mathcal{N}) \uplus (\!|e \geq 1|\!)(\mathcal{N})) \quad (46)$$

Plus précisément,

- les affectations de la forme $x = e$, $x = \mathsf{getuchar}()$ et $p = q + e$ modifient en conséquence la valeur abstraite de x et de p_o dans le polyèdre $\mathcal{N}$ grâce à l'opérateur $(\!|\cdot \leftarrow \cdot|\!)$. Par ailleurs, pour l'affectation de pointeur, la valeur abstraite de la location de p est remplacée par celle du pointeur q. La propriété de correction (36) découle directement de celle de $(\!|\cdot \leftarrow \cdot|\!)$.
- l'allocation dynamique $p = \mathsf{malloc}_\alpha(e)$ actualise la taille α_s avec la valeur abstraite de e à laquelle on a appliqué la condition $e \geq 1$ (seul ce cas-là est défini dans la sémantique concrète). Il faut par ailleurs ajouter les anciennes valeurs de α_s strictement positives, car α peut représenter plusieurs blocs mémoire distincts (mais tous alloués par $p = \mathsf{malloc}_\alpha(e)$).[14] Enfin, les valeurs de p_ℓ et p_o sont modifiées à $\{\alpha\}$ et 0 respectivement, afin que le pointeur p pointe vers le début du bloc.

[14] Seules les anciennes valeurs de α_s *strictement positives* importent, car le fait que $\nu(\alpha_s) = 0$ dans la concrétisation impose qu'aucun l_α n'apparaisse dans $\mathsf{dom}(h)$.

– pour l'affectation dans le tas $*p = e$, le seul effet est d'exclure le cas où p n'est pas initialisé (d'où $\mathcal{L}(p_\ell) \cap \mathsf{Alloc}$).
– enfin, les conditions *cond* consistent en l'application des mêmes conditions sur les valeurs numériques abstraites.

Etant donné un programme P, la fonction abstraite de transfert $\mathcal{F}$ correspondante est évidemment calculable. En appliquant le théorème 1 (section 4.2), nous en déduisons un algorithme de calcul d'une sur-approximation de la sémantique collectrice de $\mathcal{C}(P)$.

Corollaire 1 *Si l'on définit la suite* $(\mathcal{X}_n)_n$ *par :*

$$\begin{cases} \mathcal{X}_0 \stackrel{def}{=} \bot \\ \mathcal{X}_{n+1} \stackrel{def}{=} \begin{cases} \mathcal{X}_n & si\ \mathcal{F}(\mathcal{X}_n) \sqsubseteq \mathcal{X}_n \\ \mathcal{X}_n \nabla \mathcal{F}(\mathcal{X}_n) & sinon \end{cases} \end{cases} , \tag{47}$$

alors cette suite est stationnaire. Sa limite $\mathcal{X}_N$ *est* calculable *et vérifie :*

$$\mathcal{C}(P) \subseteq \gamma(\mathcal{X}_N) \tag{48}$$

Absence d'erreurs La sur-approximation de la sémantique collectrice $\mathcal{C}(P)$ d'un programme P ainsi calculée va nous permettre, lorsqu'elle est suffisamment précise, de montrer l'absence de dépassements de tampon. Ainsi, considérons la propriété suivante :

$$\forall i \in \mathsf{Ctrl}.\mathrm{si}\ \langle i, *p = e, j\rangle,\ \text{alors}\ \mathcal{X}_N(i) = (\mathcal{L}_i, \mathcal{N}_i)\ \text{vérifie}$$
$$\omega \notin \mathcal{L}_i(p) \wedge \mathcal{N}_i \subseteq \bigcap_{\alpha_s \in \mathcal{L}_i(p)} \{0 \leq p_o \leq \alpha_s - 1\}\ . \tag{49}$$

Si cette propriété est vérifiée, alors l'absence d'erreur à l'exécution de P est garantie :

Proposition 2. *Si* $\mathcal{X}_N$ *vérifie la propriété* (49)*, alors pour toute instruction* $\langle i, *p = e, j\rangle$ *du programme et pour tout état* $(i, s \mid h) \in \mathcal{C}(P)$*, la propriété* (14) *est vérifiée : le déréférencement est* sûr*.*

Cette proposition est la conséquence de l'équation (48) et de l'invariant sur les états de la mémoire (équation (1)).

Exemple 10 Appliquons maintenant le formalisme développé afin de prouver l'absence d'erreurs à l'exécution dans le programme $P_{receive}$ donné à l'exemple 1. Nous allons tout d'abord calculer la suite des $\mathcal{X}_n$ définie par le corollaire 1.

Nous partons de l'élément $\mathcal{X}_0 = \bot$:

$$\mathcal{X}_0 = \begin{cases} 1 \mapsto (\{p_\ell \mapsto \emptyset, t_\ell \mapsto \emptyset\}, \emptyset) \\ 2 \mapsto (\{p_\ell \mapsto \emptyset, t_\ell \mapsto \{\omega\}\}, \emptyset) \\ \quad \vdots \\ 12 \mapsto (\{p_\ell \mapsto \emptyset, t_\ell \mapsto \emptyset\}, \emptyset) \end{cases} .$$

Dans un souci de lisibilité, nous ne précisons pas dans la suite les valeurs associées à certains points de contrôle, lorsque celles-ci restent inchangées par rapport à l'état précédent.

L'état $\mathcal{X}_1$ est caractérisé par l'allocation dans le tas d'un nouveau bloc mémoire au site α.

$$\mathcal{X}_1 = \left\{ \begin{array}{l} 1 \mapsto (\{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\omega\}\}, \{i \geq 0, n \geq 0, sz \geq 0, tmp \geq 0, \alpha_s = 0\}) \\ \quad \vdots \end{array} \right. .$$

L'écriture d'un caractère provenant de l'extérieur dans sz laisse sa valeur abstraite inchangée : nous n'avons pas d'information précise sur ce caractère, à l'exception de son type **uchar** :

$$\mathcal{X}_2 = \left\{ \begin{array}{l} 1 \mapsto (\{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\omega\}\}, \{i \geq 0, n \geq 0, sz \geq 0, tmp \geq 0, \alpha_s = 0\}) \\ 2 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. .$$

Les états $\mathcal{X}_3$, $\mathcal{X}_4$, $\mathcal{X}_5$ et $\mathcal{X}_6$ retracent la normalisation de sz à une valeur plus petite que n (au point de contrôle 5), et les initialisations de i à 0 et de p à t :

$$\mathcal{X}_3 = \left\{ \begin{array}{l} \quad \vdots \\ 2 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ 3 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. ,$$

$$\mathcal{X}_4 = \left\{ \begin{array}{l} \quad \vdots \\ 3 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ 4 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, tmp \geq 0, t_o = 0, \alpha_s = n \leq sz - 1\} \end{array} \right) \\ 5 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. ,$$

$$\mathcal{X}_5 = \left\{ \begin{array}{l} \quad \vdots \\ 4 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, tmp \geq 0, t_o = 0, \alpha_s = n \leq sz - 1\} \end{array} \right) \\ 5 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i \geq 0, n \geq 1, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 6 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, n \geq 1, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \quad \vdots \end{array} \right. ,$$

$$\mathcal{X}_6 = \left\{ \begin{array}{l} \vdots \\ 6 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, n \geq 1, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 7 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, n \geq 1, tmp \geq 0, p_o = t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. .$$

Les états $\mathcal{X}_7$, $\mathcal{X}_8$, $\mathcal{X}_9$ et $\mathcal{X}_{10}$ correspondent notamment à une première propagation des sur-approximations dans le corps de la boucle située au point de contrôle 7 :

$$\mathcal{X}_7 = \left\{ \begin{array}{l} \vdots \\ 7 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, n \geq 1, tmp \geq 0, p_o = t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 8 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \\ 12 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, tmp \geq 0, 0 = p_o = t_o = i = sz \leq \alpha_s = n\} \end{array} \right) \end{array} \right. ,$$

$$\mathcal{X}_8 = \left\{ \begin{array}{l} \vdots \\ 8 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 9 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. ,$$

$$\mathcal{X}_9 = \left\{ \begin{array}{l} \vdots \\ 9 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 10 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. ,$$

$$\mathcal{X}_{10} = \left\{ \begin{array}{l} \vdots \\ 10 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 11 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = 1, t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. .$$

Dans l'état $\mathcal{X}_{11}$, l'état de la mémoire au point de contrôle 7 est modifié en fonction de celui du point 11. L'opérateur d'élargissement renvoie alors un état

de la mémoire dans lequel $i = p_o \leq sz \leq \alpha_s = n$:

$$\mathcal{X}_{11} = \left\{ \begin{array}{l} \vdots \\ 7 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq i = p_o \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \\ 11 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{i = 0, tmp \geq 0, p_o = 1, t_o = 0, 1 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. .$$

Les états $\mathcal{X}_{12}$, $\mathcal{X}_{13}$, $\mathcal{X}_{14}$ et $\mathcal{X}_{15}$ correspondent à une seconde propagation des états abstraits de la mémoire dans le corps de boucle :

$$\mathcal{X}_{12} = \left\{ \begin{array}{l} \vdots \\ 7 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq i = p_o \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 8 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \\ 12 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq p_o = i = sz \leq \alpha_s = n\} \end{array} \right) \end{array} \right. ,$$

$$\mathcal{X}_{13} = \left\{ \begin{array}{l} \vdots \\ 8 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 9 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. ,$$

$$\mathcal{X}_{14} = \left\{ \begin{array}{l} \vdots \\ 9 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 10 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. ,$$

$$\mathcal{X}_{15} = \left\{ \begin{array}{l} \vdots \\ 10 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 11 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o - 1 \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \vdots \end{array} \right. .$$

L'état $\mathcal{X}_{16}$ est identique à $\mathcal{X}_{15}$ puisque $\mathcal{F}(\mathcal{X}_{15}) \sqsubseteq \mathcal{X}_{15}$:

$$\mathcal{X}_{16} = \left\{ \begin{array}{l} 1 \mapsto (\{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\omega\}\}, \{i \geq 0, n \geq 0, sz \geq 0, tmp \geq 0, \alpha_s = 0\}) \\ 2 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i \geq 0, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ 3 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i \geq 0, sz \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n\} \end{array} \right) \\ 4 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i \geq 0, tmp \geq 0, t_o = 0, \alpha_s = n \leq sz - 1\} \end{array} \right) \\ 5 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i \geq 0, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 6 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\omega\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, i = 0, tmp \geq 0, t_o = 0, 0 \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 7 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{n \geq 1, t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz \leq \alpha_s = n\} \end{array} \right) \\ 8 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 9 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 10 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 11 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, tmp \geq 0, 0 \leq i = p_o - 1 \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ 12 \mapsto \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq p_o = i = sz \leq \alpha_s = n\} \end{array} \right) \end{array} \right.$$

La limite de la suite est donc atteinte.

Vérifions maintenant que la condition (49) est vérifiée par $\mathcal{X}_{16}$. Le seul déréférencement de pointeur du programme $*p = tmp$ est situé au point de contrôle 9. Nous avons :

$$\begin{array}{rl} \mathcal{X}_{16}(9) = & \left(\begin{array}{l} \{p_\ell \mapsto \{\alpha\}, t_\ell \mapsto \{\alpha\}\}, \\ \quad \{t_o = 0, n \geq 1, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, sz \leq \alpha_s = n\} \end{array} \right) \\ \stackrel{def}{=} & (\mathcal{L}_9, \mathcal{N}_9) \ , \end{array}$$

par conséquent $\omega \notin \mathcal{L}_9(p_\ell) = \{\alpha\}$, et $\mathcal{N}_9 \subseteq \{0 \leq p_o \leq \alpha_s - 1\}$. La proposition 2 nous permet donc de conclure que le programme $P_{receive}$ ne provoquera pas d'erreurs à l'exécution.

Remarquons qu'en l'absence de la normalisation de la valeur de sz à une valeur plus petite que celle de n (points de contrôle 3 et 4), le programme $P_{receive}$ pourrait provoquer un dépassement de tampon, et l'analyse aurait correctement signalé une alarme : l'invariant numérique au point de contrôle 9 aurait été $\mathcal{N}_9 = \{t_o = 0, tmp \geq 0, 0 \leq i = p_o \leq sz - 1, \alpha_s = n \geq 1\}$. Or, $\mathcal{N}_9 \not\subseteq \{0 \leq p_o \leq \alpha_s - 1\}$.

Par ailleurs, si les intervalles avaient été utilisés à la place des polyèdres comme domaine numérique abstrait (comme c'est le cas dans [31]), l'invariant

numérique en 9 aurait été $\mathcal{R}_9 = \{t_0 \mapsto [0;0]\,, tmp \mapsto [0;+\infty[\,, p_o \mapsto [0;+\infty[\,, i \mapsto [0;+\infty[\,, n \mapsto [1;+\infty[\,, \alpha_s \mapsto [1;+\infty[\,, sz \mapsto [1;+\infty[\}$, ce qui n'est pas assez précis pour assurer que $0 \leq p_o \leq \alpha_s - 1$. Un domaine plus précis, comme les polyèdres, est donc nécessaire pour montrer l'absence d'erreurs dans $P_{receive}$. ❑

5 Extensions possibles

Le formalisme tel que nous l'avons présenté dans les parties précédentes peut être enrichi de manière à couvrir de nombreuses fonctionnalités supplémentaires de langages de programmation tels que C. Nous donnons ici quelques exemples des extensions possibles.

Tout d'abord, avec très peu de modifications, des conditions plus complexes peuvent être introduites dans le langage en ajoutant la conjonction et la disjonction de conditions. La définition de la sémantique concrète de ces conditions est analogue à celle donnée à la figure 8, et leur abstraction est réalisée en utilisant les opérateurs abstraits d'union et d'intersection.

De même, la construction $x = *p$ lisant une donnée dans le tas pour la stocker dans la pile peut être intégrée. Il suffit pour cela d'ajouter la règle concrète :

$$\frac{p = (l_\alpha, o) \in \mathsf{dom}(h) \qquad h(l_\alpha, o) = v}{s \shortmid h \vdash x = *p : s[x \mapsto v] \shortmid h}\,, \tag{50}$$

et dans le modèle abstrait, de définir :

$$[\![x = *p]\!](\mathcal{L}, \mathcal{N}) = \left(\begin{array}{l} \mathcal{L}[p_\ell \mapsto \mathcal{L}(p_\ell) \cap \mathsf{Alloc}], \\ \quad (\!|x \geq 0|\!) \circ (\!|x \leftarrow ?|\!) \left(\uplus_{\alpha \in \mathcal{L}(p_\ell) \cap \mathsf{Alloc}} (\!|p_o \geq 0|\!) \circ (\!|p_o \leq \alpha_s - 1|\!)(\mathcal{N})\right) \end{array}\right)\,, \tag{51}$$

ce qui correspond à restreindre la valeur abstraite de p de manière à ce que le déréférencement soit défini, puis affecter une valeur quelconque à x (puisque nous n'avons aucune information sur la valeur de $*p$ donnée par l'abstraction du tas). Par ailleurs, la validité du déréférencement de p à la lecture peut être ajoutée dans les propriétés à vérifier.

De nombreuses constructions peuvent être ajoutées au langage, leur formalisation posant des difficultés orthogonales à celles soulevées ici : (i) d'autres types entiers, comme les booléens, les entiers signés ou non-signés, les énumérations, divers attributs (comme **short** et **long** en C), ainsi que l'analyse de dépassements de capacité dans les calculs [3], (ii) le pointeur **null**, les pointeurs de profondeur quelconque, les pointeurs vers la pile, les multiples déréférencements, les tableaux (multidimensionnels), les structures de données, les unions, les chaînes de caractères [36,1], (iii) des commandes de flots de contrôle plus élaborées, comme les appels de fonctions non-récursives, les sauts, diverses constructions de boucles, les déclarations locales de variables (voir par exemple [1]).

Enfin, le domaine coûteux des polyèdres peut être remplacés par un domaine moins précis, mais plus efficace. Par exemple, le domaine des octogones [35] (invariants numériques de la forme $\pm x \pm y \leq c$, où x et y sont des variables, c une constante) aurait suffit pour l'analyse du programme $P_{receive}$.

6 Travaux relatifs

La plupart des analyseurs statiques ne sont pas *corrects* et ne font que signaler des bogues potentiels à l'utilisateur. En d'autres termes, lorsqu'ils prétendent que le programme analysé ne provoque pas d'erreurs, on ne peut pas pour autant conclure que ce dernier est sûr. En pratique, des états atteignables de la machine sont oubliés au cours de l'analyse, souvent pour améliorer les performances en temps ou en mémoire de l'analyseur. Par conséquent, ils ne peuvent pas être utilisés pour garantir la sécurité de logiciels critiques. Les méthodes qu'ils emploient sont très variées : (i) la reconnaissance syntaxique de motifs (GNU grep [27]), (ii) avec heuristiques (flawfinder [22], ITS4 [45]), (iii) et plus généralement la propagation de propriétés, de diverses natures (numériques, interactions de pointeurs) mais sans assurance de correction (Splint [19], Coverity [14], BOON [46], EauClaire [7], CCA [6], Uno [28]). Ces outils se focalisent notamment sur la détection de variables non initialisées, de déréférencement de pointeurs **null**, d'accès illégaux dans un tableau, et d'appels à des fonctions de bibliothèques potentiellement dangereuses (comme `strcpy` en C). Les langages analysés sont, par exemple, C, C++, Java et ADA.

D'autres analyseurs statiques reposent sur des méthodes formelles *correctes*. Outre l'interprétation abstraite, citons par exemple : (i) le *model checking* [8,18], dont le principe de base est d'explorer l'ensemble des états atteignables du système (cela n'étant réalisable que lorsque le modèle à vérifier est effectivement fini, l'exploration peut être réalisée sur des abstractions finies du modèle [9]), (ii) l'abstraction par prédicats (*predicate abstraction*, [26]) est une branche du *model checking* utilisant un modèle booléen généré à partir de prédicats initialement définis, d'autres prédicats pouvant être ajoutés si nécessaire pour affiner le modèle initial [15,2], (iii) le *theorem proving* [20,21], qui traduit la sémantique du programme et les propriétés à vérifier en formules logiques, puis tente de les prouver à l'aide d'un assistant de preuve.

Les analyses utilisant le cadre formel de l'interprétation abstraite couvrent de nombreuses propriétés sur les programmes : entre autres, (i) des propriétés numériques sur les entiers (dépassement de capacité [3,38]) et sur les flottants (opérations non définies [3], précision numérique [25]), (ii) des propriétés numériques sur les pointeurs (dépassement de tampon [44,36,31]) et la longueur des chaînes de caractères [1,17], (iii) des propriétés sur la forme de la mémoire (listes et arbres [41], séparation de la mémoire en parties disjointes [16]), (iv) les invariants sur les classes dans un langage orienté objets [33], (v) des propriétés de sécurité pour les protocoles cryptographiques [4,5].

7 Conclusion

Nous avons construit une analyse statique, reposant sur l'interprétation abstraite, capable de montrer l'absence de dépassements de tampon dans un programme. En particulier, le programme donné en introduction a été analysé avec succès. Par ailleurs, plusieurs extensions possibles ont été décrites.

L'analyse statique utilisant des méthodes formelles *correctes* permet d'assurer qu'un logiciel est dépourvu de vulnérabilités. C'est un élément fondamental dans la construction d'infrastructures logicielles critiques réellement fiables.

Remerciements Merci à Alexandre Gazet et Wenceslas Godard pour leurs commentaires et leur aide dans la relecture de cet article.

Références

1. Xavier Allamigeon, Wenceslas Godard et Charles Hymans : Static Analysis of String Manipulations in Critical Embedded C Programs. *In* Kwangkeun Yi, éditeur : *Static Analysis, 13th International Symposium (SAS'06)*, volume 4134 de *Lecture Notes in Computer Science*, pages 35–51, Seoul, Korea, août 2006. Springer Verlag.
2. Thomas Ball, Byron Cook, Satyaki Das et Sriram Rajamani : Refining approximations in software predicate abstraction. *In Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 388–403. Springer-Verlag, March 2004.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux et X. Rival : A static analyzer for large safety-critical software. *In ACM SIGPLAN PLDI'03*, volume 548030, pages 196–207. ACM Press, June 2003.
4. Bruno Blanchet : An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. *In 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, juin 2001. IEEE Computer Society.
5. Bruno Blanchet : A computationally sound mechanized prover for security protocols. *In IEEE Symposium on Security and Privacy*, pages 140–154, Oakland, California, mai 2006.
6. C Code Analyzer. `http://www.drugphish.ch/~jonny/cca.html`.
7. Brian Chess : Improving computer security using extended static checking. *In SP '02 : Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 160, Washington, DC, USA, 2002. IEEE Computer Society.
8. E. M. Clarke, E. A. Emerson et A. P. Sistla : Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
9. Edmund M. Clarke, Orna Grumberg et David E. Long : Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
10. P. Cousot et R. Cousot : Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
11. P. Cousot et R. Cousot : Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.
12. P. Cousot et R. Cousot : Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

13. P. Cousot et N. Halbwachs : Automatic discovery of linear restraints among variables of a program. *In Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

14. Coverity. `http://www.coverity.com`.

15. Satyaki Das et David L. Dill : Counter-example based predicate discovery in predicate abstraction. *In Formal Methods in Computer-Aided Design*. Springer-Verlag, November 2002.

16. Dino Distefano, Peter W. O'Hearn et Hongseok Yang : A local shape analysis based on separation logic. *In* Holger Hermanns et Jens Palsberg, éditeurs : *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006*, volume 3920 de *Lecture Notes in Computer Science*, pages 287–302. Springer, mars 2006.

17. Nurit Dor, Michael Rodeh et Mooly Sagiv : Cssv : towards a realistic tool for statically detecting all buffer overflows in C. *In PLDI '03 : Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, New York, NY, USA, 2003. ACM Press.

18. Jr. Edmund M. Clarke, Orna Grumberg et Doron A. Peled : *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

19. David Evans et David Larochelle : Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, 2002.

20. Jean-Christophe Filli&#226 ;tre : Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13(4):709–745, 2003.

21. Jean-Christophe Filliâtre et Claude Marché : Multi-prover verification of C programs. *In Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 de *Lecture Notes in Computer Science*, pages 15–29. Springer Verlag, 2004.

22. Flawfinder. `http://www.dwheeler.com/flawfinder/`.

23. International Organization for Standardization : *ISO/IEC 9899 :1999 : Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, décembre 1999.

24. Jack Ganssle : Big Code. `http://www.embedded.com/showArticle.jhtml?articleID=171203287`.

25. Eric Goubault et Sylvie Putot : Static analysis of numerical algorithms. *In* Kwangkeun Yi, éditeur : *Static Analysis, 13th International Symposium (SAS'06)*, volume 4134 de *Lecture Notes in Computer Science*, pages 18–5134, Seoul, Korea, août 2006. Springer Verlag.

26. S. Graf et H. Saidi : Construction of abstract state graphs with PVS. *In* O. Grumberg, éditeur : *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

27. GNU grep. `http://www.gnu.org/software/grep/`.

28. G. J. Holzmann : Static source code checking for user-defined properties. *In Proc. IDPT 2002*, Pasadena, CA, USA, 2002.

29. Charles Hymans et Olivier Levillain : Newspeak : Big Brother is compiling your code. Rapport technique, EADS France, 2007.

30. Jason R. Ghidella et Jon Friedman : Streamlined development of body electronics systems using model-based design. http://www.mathworks.com/company/pressroom/newsletter/sept06/body_electronics.html.

31. Yungbum Jung, Jaehwang Kim, Jaeho Shin et Kwangkeun Yi : Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. *In* Igor Siveroni Chris Hankin, éditeur : *Static Analysis : 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005. Proceedings*, Lecture Notes in Computer Science, pages 203–217. Springer-Verlag, September 2005.

32. Michael Karr : Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

33. Francesco Logozzo : Automatic inference of class invariants. *In Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, volume 2937 de *Lectures Notes in Computer Science*. Springer-Verlag, janvier 2004.

34. A. Miné : A new numerical abstract domain based on difference-bound matrices. *In PADO II*, volume 2053 de *LNCS*, pages 155–172. Springer-Verlag, May 2001. http://www.di.ens.fr/~mine/publi/article-mine-padoII.pdf.

35. A. Miné : The octagon abstract domain. *In AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. http://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf.

36. A. Miné : Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. *In ACM SIGPLAN LCTES'06*, pages 54–63. ACM Press, June 2006. http://www.di.ens.fr/~mine/publi/article-mine-lctes06.pdf.

37. George C. Necula, Scott McPeak, Shree Prakash Rahul et Westley Weimer : CIL : Intermediate language and tools for analysis and transformation of C programs. *In CC '02 : Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

38. Polyspace. http://www.polyspace.com.

39. H. Gordon Rice : On completely recursively enumerable classes and their key arrays. *J. Symb. Log.*, 21(3):304–308, 1956.

40. Ron J. Pehrson : Software development for the Boeing 777. http://www.stsc.hill.af.mil/crosstalk/1996/01/Boein777.asp.

41. Shmuel Sagiv, Thomas W. Reps et Reinhard Wilhelm : Parametric shape analysis via 3–valued logic. *In Symposium on Principles of Programming Languages*, pages 105–118, 1999.

42. S. Sankaranarayanan, H. B. Sipma et Z. Manna : Scalable analysis of linear systems using mathematical programming. *In* R. Cousot, éditeur : *Verification, Model Checking and Abstract Interpretation : Proceedings of the 6th International Conference (VMCAI 2005)*, volume 3385 de *Lecture Notes in Computer Science*, pages 25–41, Paris, France, 2005. Springer-Verlag, Berlin.

43. Alfred Tarski : A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

44. Arnaud Venet et Guillaume Brat : Precise and efficient static array bound checking for large embedded C programs. *In PLDI '04 : Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 231–242, New York, NY, USA, 2004. ACM Press.

45. J. Viega, J. T. Bloch, Y. Kohno et G. McGraw : Its4 : A static vulnerability scanner for C and C++ code. *In ACSAC '00 : Proceedings of the 16th Annual Computer Security Applications Conference*, page 257, Washington, DC, USA, 2000. IEEE Computer Society.

46. David Wagner, Jeffrey S. Foster, Eric A. Brewer et Alexander Aiken : A first step towards automated detection of buffer overrun vulnerabilities. *In Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

47. Wikipedia : Source lines of code — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Lines_of_code`.