Analyses de pointeurs

TAS : Typage et analyse statique
M2, Master STL INSTA, UPMC

Antoine Miné

Année 2016–2017

Cours 13
16 mars 2017

Introduction

Nous avons considéré jusqu'à présent uniquement
des analyses numériques.

Les langages réalistes ont :

- des pointeurs (vision bas niveau : C, C++)
- des références (vision haut niveau : Java, OCaml)
- de l'allocation dynamique de mémoire
- éventuellement, de l'arithmétique de pointeur (C, C++)

Une analyse doit tenir compte des pointeurs car :

1. les pointeurs causent des erreurs spécifiques
   accès à un pointeur invalide, dépassement de capacité, double libération, etc.

   $\Longrightarrow$ nous voulons vérifier statiquement ces erreurs
2. la présence de pointeurs rend l'analyse des propriétés numériques plus difficile.
   simplement ignorer les pointeurs donne des résultats non sûrs !

Application 1 : analyse de constantes

Exemple

```
int x = 2;
int *p = &x;
(*p)++;
int y = x + 1;
```

Une analyse de constantes qui ignore les pointeurs trouverait :
`x` constant égal à 2.

Un compilateur optimisant utilisant cette analyse de constantes replacerait `y = x + 1` par `y = 3`
ce qui est faux !

Problème : un accès indirect par pointeur modifie la valeur de `x`.

C'est le problème de l'aliasing.

Application 2 : correction de manipulation mémoire en C

Exemple

```
char* p = malloc(100);
char* q = p;
for (int i=0; i<10; i++, q++) *q = 12;
free(p);
(*q)++;
```

- après `malloc`, `p` pointe vers un bloc valide ;
- après `q = p`, `q` pointe vers un bloc valide ;
- dans la boucle, `q` reste valide (i.e., dans le bloc) ;
- `free(p)` libère le bloc
  $\Longrightarrow$ `p` et `q` deviennent invalides.

Une analyse statique de pointeurs permet de détecter ces erreurs.

Application 3 : analyse de classe en Java

Exemple

```
class Rectangle {
  public void draw() { ...}
}

class Square extends Rectangle {
  @Override public void draw() { ...}
}

Rectangle x;
x = new Square();
...
x.draw();
```

L'analyse statique permet une optimisation à la compilation :

- `x` est une référence vers un objet de classe `Square`
- $\Longrightarrow$ `x.draw` peut être compilé en un appel direct à la méthode de `Square` (au lieu d'un appel indirect avec test dynamique de la classe de l'objet)

Analyse de pointeurs et analyse d'alias

Deux problèmes liés :

- analyse de pointeurs :

  étant donné un pointeur p :

  - vers quelles variables p peut-il pointer ?
  - vers quels blocs alloués dynamiquement p peut-il pointer ?

- analyse d'alias :

  étant donnés deux pointeurs p et q :

  - p et q peuvent-ils pointer vers le même bloc mémoire ?

L'analyse d'alias peut se réduire à l'analyse de pointeurs.

Note : il existe également des analyses d'alias dédiées, mais nous n'en parlerons pas.

Plan

But : montrer des exemples

1. d'analyses non-numériques : les pointeurs ;
2. d'analyses insensibles au flot de contrôle, très efficaces
   $\Longrightarrow$ utilisables dans un compilateur
   (mais moins précises que les analyses sensibles au flot de contrôle vues jusqu'à présent)

Plan :

- langage de pointeurs simple :
- sémantique concrète, non calculable ;
- abstraction par site d'allocation ;
- analyse classique, sensible au flot de contrôle ;
- analyses insensibles au flot de contrôle :
  - analyse d'Andersen ;
  - analyse de Steensgaard.

Langage

Syntaxe

Instructions

stat	::=	$X \leftarrow \mathbf{alloc}()$	(allocation)
	|	$X \leftarrow \&Y$	(prise de référence)
	|	$X \leftarrow *Y$	(lecture par référence)
	|	$*X \leftarrow Y$	(écriture par référence)
	|	$X \leftarrow expr$	(affectation numérique)
	|	$\ldots$	

- nous ajoutons aux instructions numériques des instructions de pointeur ;
- une variable peut contenir un pointeur ou un entier ;
- les manipulations de pointeurs sont réduites à des instructions atomiques ;
  nous supposons que les instructions complexes sont décomposées en instructions atomiques ;
  ainsi, `*p = *q + 1` s'écrit : $tmp \leftarrow *q; tmp \leftarrow tmp + 1; *p \leftarrow tmp$
- **alloc**() alloue une case mémoire pouvant contenir un pointeur ou un entier ;
- pas d'arithmétique de pointeur, pas de libération de mémoire.

Modélisation concrète de la mémoire

L'ajout de pointeurs nécessite d'enrichir la notion de variable et de valeur.

Adresses $\mathbb{A}$: $\mathbb{A} \stackrel{\text{def}}{=} \mathbb{V} \cup \mathbb{H}$

Dénote une portion de mémoire où une valeur peut être stockée :

- $\mathbb{V}$: les variables du programme
- $\mathbb{H}$: **le tas**, un ensemble infini d'adresses mémoires

  alloc() pioche une nouvelle adresse dans $\mathbb{H}$ quand il est exécuté

Valeurs $\mathbb{V}$: $\mathbb{V} \stackrel{\text{def}}{=} \mathbb{A} \cup \mathbb{Z}$

- l'adresse d'une variable $\mathbb{V}$
- ou l'adresse d'un bloc alloué dynamiquement $\mathbb{H}$
- ou un entier

Modélisation concrète de la mémoire (suite)

Environnements concrets : $\mathcal{E} \stackrel{\text{def}}{=} \mathbb{A} \rightharpoonup \mathbb{V}$

- associe une valeur à une adresse mémoire
- $\rightharpoonup$ dénote une fonction partielle
  toutes les adresses ne sont pas allouées !
- si $\rho \in \mathcal{E}$, $dom(\rho) \subseteq \mathbb{A}$ dénote les adresses effectivement allouées
  - $\rho(v)$ est indéfini si $v \notin dom(\rho)$
  - $dom(\rho)$ est un ensemble fini
    mais $\mathbb{A}$ est infini, pour modéliser des programmes utilisant une mémoire arbitraire
  - $\mathbb{V} \subseteq dom(\rho)$: les variables existent toujours en mémoire

Sémantique concrète

$\mathsf{S}[\![\, stat \,]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$

$$
\begin{array}{lcl}
\mathsf{S}[\![\, X \leftarrow \mathbf{alloc}() \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho[X \mapsto \ell, \ell \mapsto 0] \mid \rho \in R, \ell \notin dom(\rho) \,\} \\
\mathsf{S}[\![\, X \leftarrow \&Y \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho[X \mapsto Y] \mid \rho \in R \,\} \\
\mathsf{S}[\![\, X \leftarrow Y \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho[X \mapsto \rho(Y)] \mid \rho \in R \,\} \\
\mathsf{S}[\![\, X \leftarrow *Y \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho[X \mapsto \rho(\rho(Y))] \mid \rho \in R, \rho(Y) \in dom(\rho) \,\} \\
\mathsf{S}[\![\, *X \leftarrow Y \,]\!]\, R & \stackrel{\text{def}}{=} & \{\, \rho[\rho(X) \mapsto \rho(Y)] \mid \rho \in R, \rho(X) \in dom(\rho) \,\}
\end{array}
$$

Notes :

- $\rho[\ell \mapsto x], \ell \notin dom(\rho)$ enrichit $dom(\rho)$
- les blocs alloués sont initialisés à 0
- $X \mapsto Y$ stocke l'adresse de la variable Y dans la variable X
- $X \mapsto \rho(Y)$ stocke la valeur de la variable Y dans la variable X
- $X \mapsto \rho(\rho(Y))$ stocke la valeur pointée par Y dans la variable X
- $\rho(X) \mapsto v$ stocke v à l'adresse pointée par la variable X
- $\rho(Y) \in dom(\rho)$ signifie : Y pointe sur une adresse valide
- pas de modification à $\mathsf{S}[\![\, X \leftarrow expr \,]\!]$, $\mathsf{C}[\![\, \cdot \,]\!]$

  car les expressions ne contiennent que des variables,
  il faut simplement vérifier que $\rho(V) \notin \mathbb{A}$ dans les expressions

Analyse statique

Principe d'analyse

Principe : décomposition du problème

1. analyse de pointeurs : $ptr : \mathbb{V} \to \mathcal{P}(\mathbb{A})$

   déterminer où chaque variable peut pointer

2. utiliser l'information de l'analyse pour éliminer les pointeurs du programme par transformation de programme

   Exemple :

```
int x = 2;                          int x = 2;
int *p = &x;        ptr(p)={x}
(*p)++;             ⟹              x++;
int y = x + 1;                      int y = x + 1;
```

3. analyse numérique : dans $\mathcal{P}(\mathbb{A} \to \mathbb{Z})$

   réutilisation des domaines numériques présentés en cours

Note :

une analyse combinée de pointeurs et d'entiers est possible, avec un produit réduit ; elle serait plus précise, mais un peu plus complexe à mettre en œuvre.

Affectation faible

Exemple

$X \leftarrow 1; Y \leftarrow 2;$
if $\cdots$ **then** $P \leftarrow \&X$ **else** $P \leftarrow \&Y;$
$Z \leftarrow *P;$
$*P \leftarrow 3$

Une analyse de pointeurs donnerait $ptr(P) = \{X, Y\}$.

Comment interpréter $Z \leftarrow *P$ et $*P \leftarrow 3$?
Indétermination sur $*P \Longrightarrow$ il faut traiter tous les cas !

Exemple : dans les domaines non-relationnels (e.g., intervalles)

- $\mathrm{S}^\sharp[\![Z \leftarrow \{X, Y\}]\!] R^\sharp \overset{\text{def}}{=} R^\sharp[Z \mapsto R^\sharp(X) \cup^\sharp R^\sharp(Y)]$
  union abstraite $\cup^\sharp$ de toutes les variables lues
- $\mathrm{S}^\sharp[\![\{X, Y\} \leftarrow e]\!] R^\sharp \overset{\text{def}}{=}$
  $R^\sharp[X \mapsto R^\sharp(X) \cup^\sharp \mathrm{E}^\sharp[\![e]\!] R^\sharp, Y \mapsto R^\sharp(Y) \cup^\sharp \mathrm{E}^\sharp[\![e]\!] R^\sharp]$
  - toutes la variables de $ptr(P)$ sont mises à jour ;
  - une variable peut retenir sa valeur précédente $(V \mapsto R^\sharp(V) \cup^\sharp \ldots)$

$\Longrightarrow$ les intervalles donnent : $Z \in [1, 2]$, $X \in [1, 3]$, $Y \in [2, 3]$.

Difficulté : allocation non bornée

Exemple

$X \leftarrow \textbf{rand}(0, +\infty)$;
while $i < X$ **do**
    $P \leftarrow \textbf{alloc}()$;
    $*P \leftarrow i$

Rappel : $\mathbb{A}$ est infini.

nécessaire pour donner un sens à un programme avec allocation non bornée

$\Longrightarrow$ $ptr(V)$ peut être infini !

or, l'analyse numérique ne peut se faire que sur un nombre fini de variables...

Solution : abstraction par site d'allocation

Exemple

$X \leftarrow \textbf{rand}(0, +\infty)$;
while $i < X$ **do**
    $P \leftarrow \textbf{alloc}^{\ell}()$;
    $*P \leftarrow i$

Solution : abstraire $\mathbb{A}$ en un ensemble fini $\mathbb{A}^\sharp$ d'adresses abstraites

$\mathbb{A}^\sharp$ contient :

- les variables $\mathbb{V}$
- une adresse abstraite $\ell^\sharp$
  pour chaque instruction **alloc**() du programme
  i.e., chaque site d'allocation

  $\ell^\sharp \in \mathbb{A}$ représente l'ensemble de tous les blocs alloués par $\textbf{alloc}^{\ell}()$

Exemple : $\mathbb{A}^\sharp = \{X, i, P, \ell^\sharp\}$

Avantage : $\mathbb{A}^\sharp$ est maintenant fini

Inconvénient : tout accès à $\ell^\sharp$ génère une affectation faible (perte de précision)

Analyse abstraite de pointeurs

Domaine abstrait : $\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathbb{A}^\sharp \to \mathcal{P}(\mathbb{A}^\sharp)$

- utilisation des adresses abstraites : $\mathbb{A}^\sharp$
- inférence des informations de pointeur uniquement : $\mathcal{P}(\mathbb{A}^\sharp)$
  pas de propriété numérique !

$\mathsf{S}^\sharp[\![\, stat \,]\!] : \mathcal{E}^\sharp \to \mathcal{E}^\sharp$

$$\mathsf{S}^\sharp[\![\, X \leftarrow \mathbf{alloc}^\ell() \,]\!]\, R^\sharp \stackrel{\text{def}}{=} R^\sharp[X \mapsto \{\ell^\sharp\}]$$

$$\mathsf{S}^\sharp[\![\, X \leftarrow \&Y \,]\!]\, R \stackrel{\text{def}}{=} R^\sharp[X \mapsto \{Y\}]$$

$$\mathsf{S}^\sharp[\![\, X \leftarrow *Y \,]\!]\, R \stackrel{\text{def}}{=} R^\sharp[X \mapsto R^\sharp(Y)]$$

$$\mathsf{S}^\sharp[\![\, *X \leftarrow Y \,]\!]\, R \stackrel{\text{def}}{=}$$

$$\begin{cases} R^\sharp[p \mapsto R^\sharp(Y)] & \text{si } R^\sharp(X) = \{p\} \quad \text{(affectation forte)} \\ R^\sharp[\forall p \in R^\sharp(X) : p \mapsto R^\sharp(p) \cup \{Y\}] & \text{si } |R^\sharp(X)| > 1 \quad \text{(affectation faible)} \end{cases}$$

Sémantique abstraite des manipulations de pointeurs :

- analyse non-relationnelle : abstraction indépendante de chaque variable ;
- utilisation d'affectations faibles en cas d'indétermination sur la destination.

Analyse abstraite de pointeurs (suite)

$\mathsf{S}^\sharp[\![\, stat \,]\!] : \mathcal{E}^\sharp \to \mathcal{E}^\sharp$

$$
\begin{array}{lcl}
\mathsf{S}^\sharp[\![\, s_1 ; s_2 \,]\!]\, R^\sharp & \stackrel{\text{def}}{=} & \mathsf{S}^\sharp[\![\, s_2 \,]\!]\,(\mathsf{S}^\sharp[\![\, s_1 \,]\!]\, R^\sharp) \\
\mathsf{S}^\sharp[\![\, X \leftarrow expr \,]\!]\, R^\sharp & \stackrel{\text{def}}{=} & R^\sharp \\
\mathsf{S}^\sharp[\![\, \textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2 \,]\!]\, R^\sharp & \stackrel{\text{def}}{=} & \mathsf{S}^\sharp[\![\, s_1 \,]\!]\, R^\sharp \mathbin{\dot\cup} \mathsf{S}^\sharp[\![\, s_2 \,]\!]\, R^\sharp \\
\mathsf{S}^\sharp[\![\, \textbf{while } c \textbf{ do } s \,]\!]\, R^\sharp & \stackrel{\text{def}}{=} & \mathrm{lfp}\, \lambda X^\sharp . R^\sharp \mathbin{\dot\cup} \mathsf{S}^\sharp[\![\, s \,]\!]\, X^\sharp
\end{array}
$$

Sémantique abstraite du fragment sans pointeur :

- interprétation par induction sur la syntaxe ;
- $\dot\cup$ est l'union point à point : $(R^\sharp \mathbin{\dot\cup} S^\sharp)(X) = R^\sharp(X) \cup S^\sharp(X)$;
- les affectations numériques sont ignorées ;
- les tests numériques sont interprétés comme des choix non-déterministes ;
- $\mathcal{E}^\sharp$ a une hauteur finie
  $\Longrightarrow$ les boucles calculent des itérations abstraites sans élargissement.

Note : analyses uniformes et non-uniformes

Exemple

$X \leftarrow \textbf{rand}(0, +\infty)$;
while $i < X$ **do**
    $P \leftarrow \textbf{alloc}^{\ell}()$;
    $*P \leftarrow i$

L'analyse avec notre abstraction $\mathbb{A}^\sharp \stackrel{\text{def}}{=} \{ X, i, P, \ell^\sharp \}$ donnera : $\ell^\sharp \in [0, +\infty]$

L'analyse est uniforme : elle ne distingue pas les différents blocs alloués en ℓ

Analyse non-uniforme : capable de distinguer les instances de $\textbf{alloc}^{\ell}()$

Par exemple : le $i-$ème bloc alloué contient la valeur i

Principe :

- ajouter un compteur d'allocation a : $\ell[a]$ est le $a-$ième bloc alloué en ℓ
- encoder la séquence $\ell[a]$ par une paire de variables $a, \ell^\sharp$
- utiliser un domaine relationnel pour relier $a, \ell^\sharp$ et les autres variables

$R \in \mathcal{P}(\{a, \ell^\sharp, X\} \to \mathbb{Z})$ représente les environnements ρ tels que :
$\forall a: ((\rho(a), \rho(\ell[\rho(a)]), \rho(X)) \in R$

$\Longrightarrow$ la propriété d'intérêt peut être représentée par $\ell^\sharp = a \wedge a \in [0, X]$

Analyses insensibles au flot de contrôle

Principe

Toutes les analyses que nous avons vues sont sensibles au flot de contrôle :

- elle distinguent l'état abstrait en fonction du point de programme
  $\Longrightarrow$ précis
- elle procèdent par induction sur la syntaxe du programme
  en itérant les boucles
  $\Longrightarrow$ coûteux

Certaines applications, comme la compilation optimisante, nécessitent des analyse moins coûteuses (même si elles sont moins précises)

Analyses insensibles au flot de contrôle :

- manipule un unique environnement abstrait global
  valable en tout point du programme ;
- permet un ordre arbitraire de traitement des instructions du programme (l'analyse ne suit pas forcément le flot de contrôle)

Analyse d'Andersen

Principe :

- associer à chaque variable un ensemble d'adresses abstraites, $ptr : V \to \mathbb{A}^\sharp$ à déterminer
- extraire de chaque affectation une contrainte d'inclusion sur ptr
- résoudre l'ensemble des contraintes extraites pour inférer ptr

l'instruction :	est représentée par la contrainte :
$X \leftarrow \mathbf{alloc}^\ell()$	$\{\ell\} \subseteq ptr(X)$
$X \leftarrow \&Y$	$\{Y\} \subseteq ptr(X)$
$X \leftarrow Y$	$ptr(Y) \subseteq ptr(X)$
$X \leftarrow *Y$	$\forall v \in ptr(Y)\colon ptr(v) \subseteq ptr(X)$
$*X \leftarrow Y$	$\forall v \in ptr(X)\colon ptr(Y) \subseteq ptr(v)$

- les constructions de tests et de boucles sont ignorées
  seules les affectations contenues dans ces construction sont extraites
- mélange de contraintes d'inclusion simples (facile à traiter)
  et de contraintes quantifiées (plus complexes)

Résolution des contraintes d'inclusion simples

Exemple :

programme :	contraintes :
$P \leftarrow \&X;$	$\{X\} \subseteq ptr(P)$
$Q \leftarrow P;$	$ptr(P) \subseteq ptr(Q)$
$P \leftarrow \mathbf{alloc}^{\ell}();$	$\{\ell\} \subseteq ptr(P)$
$R \leftarrow P$	$ptr(P) \subseteq ptr(R)$

La plus petite solution est : $ptr(P) = ptr(Q) = ptr(R) = \{X, \ell\}$.

Note : perte de précision, en réalité Q ne peut pas pointer sur ℓ, et R ne peut pas pointer sur X

La résolution peut se faire par un algorithme de graphe :

- un nœud pour chaque adresse $a \in \mathbb{A}^\sharp$, représentant l'ensemble $\{a\}$;
- un nœud pour chaque pointeur à déterminer : $ptr(P)$;
- un arc $v \to w$ pour chaque contrainte $v \subseteq w$;
- calcul de la clôture transitive
  $ptr(P) = \{\, a \in \mathbb{A}^\sharp \mid \text{la clôture transitive a un arc } a \to ptr(P) \,\}$

$\Longrightarrow$ coût cubique

Résolution des contraintes complexes

Exemple :

programme :	contraintes :
$P \leftarrow \&X;$	$\{X\} \subseteq ptr(P)$
$Q \leftarrow P;$	$ptr(P) \subseteq ptr(Q)$
$*P \leftarrow Q;$	$\forall v \in ptr(P)\colon ptr(Q) \subseteq ptr(v)$
$P \leftarrow \&Y;$	$\{Y\} \subseteq ptr(P)$
$S \leftarrow *P;$	$\forall v \in ptr(P)\colon ptr(v) \subseteq ptr(S)$

Principe de résolution :

1. création d'un graphe pour les contraintes simples d'inclusion
2. calcul de la clôture transitive
   donnant une première approximation de ptr
3. ajout d'arcs $v \to w$ par interprétation des contraintes complexes
   e.g., substituer la valeur trouvée de $ptr(P)$ dans $\forall v \in ptr(P)\colon ptr(Q) \subseteq ptr(v)$
4. reprendre l'étape 2, et itérer 2 et 3 jusqu'à stabilisation

Le coût est toujours cubique.

Pour notre exemple, nous trouvons :
$ptr(P) = ptr(Q) = ptr(X) = ptr(Y) = \{X, Y\}$ (création de cycle sur X et Y)

Analyse de Steensgaard

Principe :

Remplacer les contraintes d'inclusion par des contraintes d'égalité.

l'instruction :	est représentée par la contrainte :
$X \leftarrow \mathbf{alloc}^{\ell}()$	$\{\ell\} \subseteq ptr(X)$
$X \leftarrow \&Y$	$\{Y\} \subseteq ptr(X)$
$X \leftarrow Y$	$ptr(Y) = ptr(X)$
$X \leftarrow *Y$	$\forall v \in ptr(Y)\colon ptr(v) = ptr(X)$
$*X \leftarrow Y$	$\forall v \in ptr(X)\colon ptr(Y) = ptr(c)$

La clôture transitive d'une relation d'égalité est bien plus efficace à calculer que la clôture transitive d'une relation d'inclusion !

coût quasi-linéaire avec une structure de données de type *union-find*

Mais l'analyse est moins précise.

Comparaison des analyses de Steensgaard et d'Andersen

Exemple :

programme	Andersen	Steensgaard
$P \leftarrow \&X;$	$\{X\} \subseteq ptr(P)$	$\{X\} \subseteq ptr(P)$
$Q \leftarrow \&Y;$	$\{Y\} \subseteq ptr(Q)$	$\{Y\} \subseteq ptr(Q)$
$Q \leftarrow P$	$ptr(P) \subseteq ptr(Q)$	$ptr(P) = ptr(Q)$

L'analyse d'Andersen trouve : $ptr(P) = \{X\}$, $ptr(Q) = \{X, Y\}$,

L'analyse de Steensgaard trouve : $ptr(P) = \{X, Y\}$, $ptr(Q) = \{X, Y\}$

Niveaux de sensibilité

Outre le choix entre sensible et insensible au flot de contrôle, plusieurs choix de sensibilité permettent de mieux contrôler le rapport coût / précision :

- sensibilité au contexte d'appel

  en présence de procédures, distinguer les sites d'appel,
  traiter les blocs alloués comme des variables abstraites distinctes dans $\mathbb{A}^\sharp$

  Exemple :

  ```
  void* f() { return malloc(100); } a = f(); b = f();
  ```
  les blocs `a` et `b` seront abstraits séparément

- sensibilité au champ

  en présence de structures,
  chaque champ a une variable abstraite distincte dans $\mathbb{A}^\sharp$
  plus précis mais plus coûteux qu'avec une variable abstraite par structure

- des variantes mélangeant Steensgaard et Andersen existent également
  - contraintes d'inclusion pour les pointeurs de surface
    variables qui ne sont pas référencées par d'autres pointeurs
  - contraintes d'égalité pour les pointeurs profonds

  traitement précis du passage par référence sans compromettre l'efficacité

Abstractions de tableaux

Exemple

Exemple : séquence croissante

$p[0] \leftarrow 0; B[0] \leftarrow A[0];$
$i \leftarrow 1; k \leftarrow 1;$
while $i < N$ **do**
  if $A[i] > B[k-1]$ **then**
    $B[k] \leftarrow A[i];$
    $p[k] \leftarrow i;$
    $k \leftarrow k + 1;$
  $i \leftarrow i + 1$

Étant donné un tableau $A[0], \ldots, A[N-1]$
le programme extrait un sous-tableau croissant $B[0], \ldots, B[k-1]$
et les indices correspondant $p[0], \ldots, p[k-1]$

Invariants : $1 \leq k \leq i \leq N$ $\quad \forall x < k : B[x] = A[p[x]]$
$\forall x : 0 \leq p[x] < N$ $\quad \forall x < k-1 : B[x+1] > B[x]$

Plan

- Syntaxe et sémantique concrète
- Sémantique abstraite non-relationnelle

  e.g., $\forall i: A[i] \leq \textit{constante}$
  - application à l'analyse d'intervalle
- Sémantique abstraite relationelle (mais uniforme)

  e.g., $\forall i: A[i] \leq V$
  - opérations *expand* et *fold*
  - application à l'analyse dans les polyèdres
- Abstractions non-uniformes

  e.g., $\forall i: A[i] \leq i$

Extension de la syntaxe

Nouvelles expressions et instructions

$expr$	$::=$	V	accès scalaire, $V \in \mathbb{V}$
	$\mid$	$A[expr]$	accès de tableau, $A \in \mathbb{A}$
	$\mid$	$\dots$	
$stat$	$::=$	$V \leftarrow expr$	affectation scalaire, $V \in \mathbb{V}$
	$\mid$	$A[expr] \leftarrow expr$	affectation de tableau, $A \in \mathbb{A}$
	$\mid$	$\dots$	

Le langage a maintenant deux manières d'accéder à la mémoire :

- $\mathbb{V}$: variables scalaires entières (comme avant)
- $\mathbb{A}$: des tableaux d'entiers (nouveaux)
  - les tableaux sont indicés par des entiers positifs
  - les tableaux sont supposés non bornés
    (pour simplifier, nous ignorons les dépassement)

$\Longrightarrow$ un tableau A est similaire à une fonction ou table $A : \mathbb{N} \to \mathbb{Z}$

Sémantique concrète

Environnements concrets : $\mathcal{E} \stackrel{\text{def}}{=} (\mathbb{V} \cup (\mathbb{A} \times \mathbb{N})) \to \mathbb{Z}$

$\rho \in \mathcal{E}$ affecte une valeur entière à chaque "cellule mémoire" :

- $\rho(V)$ pour chaque variable scalaire $V \in \mathbb{V}$ et
- $\rho(A, i)$ pour chaque case de tableau $A \in \mathbb{A}$, $i \geq 0$

Sémantique concrète :

$$
\begin{array}{lcl}
\mathrm{E}[\![V]\!]\,\rho & \stackrel{\text{def}}{=} & \{\rho(V)\} \\
\mathrm{E}[\![A[e]]\!]\,\rho & \stackrel{\text{def}}{=} & \{\, \rho(A, i) \mid i \in \mathrm{E}[\![e]\!]\,\rho \,\} \\
\mathrm{S}[\![V \leftarrow e]\!]\,R & \stackrel{\text{def}}{=} & \{\, \rho[V \mapsto v] \mid \rho \in R,\ v \in \mathrm{E}[\![e]\!]\,\rho \,\} \\
\mathrm{S}[\![A[f] \leftarrow e]\!]\,R & \stackrel{\text{def}}{=} & \{\, \rho[(A, i) \mapsto v] \mid \rho \in R,\ v \in \mathrm{E}[\![e]\!]\,\rho,\ i \in \mathrm{E}[\![f]\!]\,\rho,\ i \geq 0 \,\}
\end{array}
$$

...

Abstractions non-relationelles

Abstraction de repliage

But : réutiliser les domaines numériques abstraits existants

Problème : les domaines numériques n'abstraient que des sous-ensembles de $\mathbb{Z}^n$, pour n fini

Solution : réduire $\mathcal{E}$ à des fonctions sur des ensembles finis de variables abstraites

Variables abstraites : $\mathbb{V}^\sharp \stackrel{\text{def}}{=} \mathbb{V} \cup \mathbb{A}$

- les variables scalaires de $\mathbb{V}$ sont exactement représentées dans $\mathbb{V}^\sharp$
- le contenu d'un taleau $A \in \mathbb{A}$ est abstrait par une unique variable de résumé A qui modélise le contenu de tout le tableau
- $\mathbb{V}^\sharp$ est fini

Correspondance de Galois de repliage :

$$(\mathcal{P}(\mathcal{E}), \subseteq) \underset{\alpha_s}{\overset{\gamma_s}{\leftrightarrows}} (\mathcal{P}(\mathbb{V}^\sharp \to \mathbb{Z}), \subseteq)$$

- $\alpha_s(R) \stackrel{\text{def}}{=} \{ [V \mapsto \rho(V), A \mapsto \rho(A, \iota(A))] \mid \rho \in R, \iota \in \mathbb{A} \to \mathbb{N} \}$
  replie tous les éléments de tableau (A, i) dans la variable abstraite A
- $\gamma_s(S) \stackrel{\text{def}}{=} \{ \rho \mid \forall \iota \in \mathbb{A} \to \mathbb{N}: [V \mapsto \rho(V), A \mapsto \rho(A, \iota(A))] \in S \}$
  on a bien : $\gamma_s(S) = \{ \rho \mid \alpha_s(\{\rho\}) \subseteq S \} = \cup \{ R \mid \alpha_s(R) \subseteq S \}$

Abstractions non-relationnelle

Rappel : abstraction d'intervalle

- $\mathcal{P}(\mathbb{V}^\sharp \to \mathbb{Z})$ est abstrait en $\mathbb{V}^\sharp \to \mathcal{P}(\mathbb{Z})$ (abstraction Cartésienne)
- $\mathcal{P}(\mathbb{Z})$ est abstrait en un intervalle, dans $\mathbb{I}$

Note : les abstractions Cartésienne et de repliage commutent

Sémantique abstraite : in $\mathcal{E}^\sharp \stackrel{\text{def}}{=} \mathbb{V}^\sharp \to \mathbb{I}$

- $\mathrm{E}^\sharp[\![V]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp(V)$
  $\mathrm{E}^\sharp[\![A[e]]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp(A)$ (e est ignoré)
- $\mathrm{S}^\sharp[\![V \leftarrow e]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp[V \mapsto \mathrm{E}^\sharp[\![e]\!] X^\sharp]$
  $\mathrm{S}^\sharp[\![A[f] \leftarrow e]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp[A \mapsto X^\sharp(A) \cup^\sharp \mathrm{E}^\sharp[\![e]\!] X^\sharp]$
  (f est ignoré $\to$ utilisation d'une affectation faible qui accumule les valeurs)
- si $X^\sharp(V) = X^\sharp(A) = [a, b]$:
  $\mathrm{S}^\sharp[\![V \leq c]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp[V \mapsto [a, \min(b, c)]]$ si $a \leq c$, $\bot$ sinon
  $\mathrm{S}^\sharp[\![A[e] \leq c]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp$ si $a \leq c$, $\bot$ sinon
  (test de satisfiabilité, mais pas de raffinement de la valeur de $X^\sharp(A)$
  le cas $A[e] \leq A[f]$ est similaire)
- les autres opérations restent inchangées : $\cap^\sharp$, $\cup^\sharp$, ...

Exemple d'analyse sur les intervalles

Exemple : sous-séquence croissante

$p[0] \leftarrow 0; B[0] \leftarrow A[0];$
$i \leftarrow 1; k \leftarrow 1;$
while $i < N$ **do**
  if $A[i] > B[k-1]$ **then**
    $B[k] \leftarrow A[i];$
    $p[k] \leftarrow i;$
    $k \leftarrow k + 1;$
  $i \leftarrow i + 1$

Résultat :

En supposant $N \in [N_\ell, N_h]$, $\forall x: A[x] \in [A_\ell, A_h]$, on trouve :

- $\forall x: p[x] \in [0, N_h - 1]$
- $\forall x: B[x] \in [\min(0, A_\ell), \max(0, A_h)]$

Abstraction relationnelle

Ajout et suppression de variable

Sémantique concrète :

L'ensemble $\mathbb{V}$ des variables varie lors de l'exécution du programme
(e.g., variables locales)

maintenant : $\mathcal{E} \stackrel{\text{def}}{=} \bigcup_{\mathbb{V} \text{ finite}} \mathbb{V} \to \mathbb{Z}$

- $\mathrm{S}[\![\textbf{add } V]\!] R \stackrel{\text{def}}{=} \{ \rho[V \mapsto v] \mid \rho \in R, v \in \mathbb{Z} \}$
  (ajout d'une variable non-initialisée)
- $\mathrm{S}[\![\textbf{del } V]\!] R \stackrel{\text{def}}{=} \{ \rho_{|_{\mathrm{dom}(\rho) \setminus \{V\}}} \mid \rho \in R \}$
  (supression d'une variable)

Sémantique abstraite :

$\mathcal{E}^\sharp \stackrel{\text{def}}{=} \bigcup_{\mathbb{V} \text{ finite}} \mathcal{E}^\sharp_{\mathbb{V}}$
(un domaine abstrait sur $|\mathbb{V}|$ dimensions pour chaque $\mathbb{V}$ possible, e.g. : $\mathcal{E}^\sharp_{\mathbb{V}}$ = polyèdres de $\mathbb{R}^{|\mathbb{V}|}$)

Exemple, dans les intervalles :

- $\mathrm{S}^\sharp[\![\textbf{add } V]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp[V \mapsto [-\infty, +\infty]]$
- $\mathrm{S}^\sharp[\![\textbf{del } V]\!] X^\sharp \stackrel{\text{def}}{=} X^\sharp_{|_{\mathrm{dom}(X^\sharp) \setminus \{V\}}}$

Duplication et repliage de variabes

expand et *fold* : modélisent le repliage dynamique

$$\mathsf{S}[\![\,\textbf{expand}\ V \to V'\,]\!]\,R \stackrel{\text{def}}{=} \{\,\rho[V' \mapsto v] \mid \rho \in R \wedge \rho[V \mapsto v] \in R\,\}$$

$$\mathsf{S}[\![\,\textbf{fold}\ V \hookleftarrow V'\,]\!]\,R \stackrel{\text{def}}{=} \{\,\rho \mid \exists v\colon \rho[V' \mapsto v] \in R \vee \rho[V' \mapsto \rho(V), V \mapsto v] \in R\,\}$$

- *expand* copie une variable et ses contraintes
  $(1 \leq V \leq X \Longrightarrow 1 \leq V \leq X \wedge 1 \leq V' \leq X$; mais $V = V'$ n'est pas vrai !)
- *fold* replie V et V' dans V
  $(1 \leq V \leq X \wedge 2 \leq V' \leq Y \Longrightarrow 1 \leq V \leq X \vee 2 \leq V \leq Y)$
- fold est une abstraction, expand est la concretization associée :

$$\mathcal{P}(\mathbb{V} \to \mathbb{Z}) \underset{\mathsf{S}[\![\,\textbf{fold}\ V \hookleftarrow V'\,]\!]}{\overset{\mathsf{S}[\![\,\textbf{expand}\ V \to V'\,]\!]}{\leftrightarrows}} \mathcal{P}((\mathbb{V} \backslash \{V'\}) \to \mathbb{Z})$$

(on a une correspondance de Galois)

Expand et *fold* relationnel

Abstraction dans les polyèdres :

- expand peut être modélise exactement par des simples copies de contraintes :

  $\mathsf{S}^\sharp[\![\,\textbf{expand}\ V_a \to V_b\,]\!]\,\{\,\sum_i \alpha_{ij} V_i \geq \beta_j\,\} \stackrel{\text{def}}{=}$

  $\{\,\sum_i \alpha_{ij} V_i \geq \beta_j\,\} \cup \{\,\sum_{i\neq a} \alpha_{ij} V_i + \alpha_{aj} V_b \geq \beta_j\,\}$

- fold peut être sur-approximé par une copie faible :

  $\mathsf{S}^\sharp[\![\,\textbf{fold}\ V \hookleftarrow V'\,]\!]\,X^\sharp \stackrel{\text{def}}{=} \mathsf{S}^\sharp[\![\,\textbf{del}\ V'\,]\!]\,(X^\sharp \cup^\sharp \mathsf{S}^\sharp[\![\,V \leftarrow V'\,]\!]\,X^\sharp)$

  (l'affectation garde les anciennes valeurs, en plus des nouvelles, au lieu de les remplacer)

  exemple : $0 \leq V \leq 3 \wedge 10 \leq V' \leq 13 \Longrightarrow 0 \leq V \leq 13$
  qui sur-approxime $0 \leq V \leq 3 \vee 10 \leq V \leq 13$

- $\mathsf{S}^\sharp[\![\,\textbf{add}\ V\,]\!]$ garde les contraintes inchangées
- $\mathsf{S}^\sharp[\![\,\textbf{del}\ V\,]\!]$ projette V

Abstraction relationnelle des tableaux

But : abstraite $\mathcal{P}(\mathcal{E})$ par des polyèdres sur $\mathbb{V}^\sharp \stackrel{\text{def}}{=} \mathbb{V} \cup \mathbb{A}$

Principe : utiliser *join* et *expand* en interne

Abstraction de l'affectation : $\mathrm{S}^\sharp[\![A[f] \leftarrow e]\!] X^\sharp$

- un accès de tableau $A[expr]$ dans e est remplacé par une copie fraîche de A ce qui donne une nouvelle experssion e' et un nouvel environnement $X_1^\sharp$

  e.g., remplacer $B[expr]$ dans $X^\sharp$, parB' dans $X_1^\sharp \stackrel{\text{def}}{=} \mathrm{S}^\sharp[\![\textbf{expand } B \to B']\!] X^\sharp$

- création d'une copie A' de A, pour contenir le résultat

  $X_2^\sharp \stackrel{\text{def}}{=} \mathrm{S}^\sharp[\![\textbf{expand } A \to A']\!] X_1^\sharp$

- affectation de e' dans A'

  $X_3^\sharp \stackrel{\text{def}}{=} \mathrm{S}^\sharp[\![A' \leftarrow e']\!] X_2^\sharp$

- repliage de A' dans A

  $X_4^\sharp \stackrel{\text{def}}{=} \mathrm{S}^\sharp[\![\textbf{fold } A \hookleftarrow A']\!] X_3^\sharp$

- suppression des copies de tableau :

  $\mathrm{S}^\sharp[\![\textbf{del } B']\!] X_4^\sharp$

($\mathrm{S}^\sharp[\![V \leftarrow e]\!]$ et $\mathrm{S}^\sharp[\![c?]\!]$ se traitent de la manière similaire)

Exemple d'analyse dans les polyèdres

Exemple : sous-séquence croissante

$p[0] \leftarrow 0; B[0] \leftarrow A[0];$
$i \leftarrow 1; k \leftarrow 1;$
while $i < N$ **do**
  if $A[i] > B[k-1]$ **then**
    $B[k] \leftarrow A[i];$
    $p[k] \leftarrow i;$
    $k \leftarrow k + 1;$
  $i \leftarrow i + 1$

Résultats :

Avec l'hypothèse $\forall x: A[x] \in [A_\ell, A_h]$, on trouve :

- $\forall x: 0 \leq p[x] < N$
  (plus fort que $\forall k: 0 \leq p[k] < N_h$)
- $\forall x: B[x] \in [\min(0, A_\ell), \max(0, A_h)]$
  (Note : $B \leq A$ n'est pas vrai, car il serait interprété comme $\forall i, j: B[i] \leq A[j]$)

Abstractions non-uniformes

Au-delà des abstractions uniformes

Le repliage $\alpha_s : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathbb{V}^\sharp \to \mathbb{Z})$ est une abstraction uniforme :
elle oublie les relations entre les indices et les valeurs des éléments de tableaux.

Exemple d'abstraction non-uniforme : segmentation de tableau

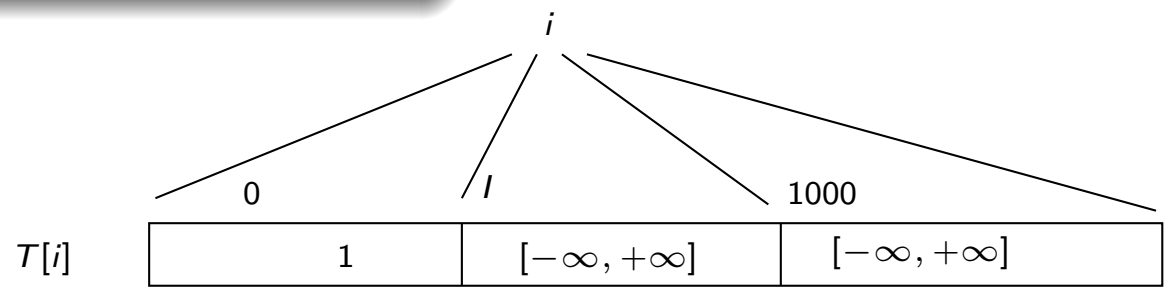
Boucle d'initialisation

$I \leftarrow 0$;
while • $I < 1000$ **do**
  $T[I] \leftarrow 1$;
  $I \leftarrow I + 1$

nous voulons analyser la boucle sans déroulement

en •, il faut exprimer l'invariant de boucle :
$\forall i < I : T[i] = 1$

$\Longrightarrow$ en sortie de boucle, T est initialisé jusqu'à 1000



domaine abstrait :

partitionne le tableau en segments uniformes
un segment a des bornes constantes ou symboliques (0, I, 1000,...)
les segments ont un contenu, exprimé dans un domaine abstrait (intervalles, ...)