Analyse statique de programmes et interprétation abstraite

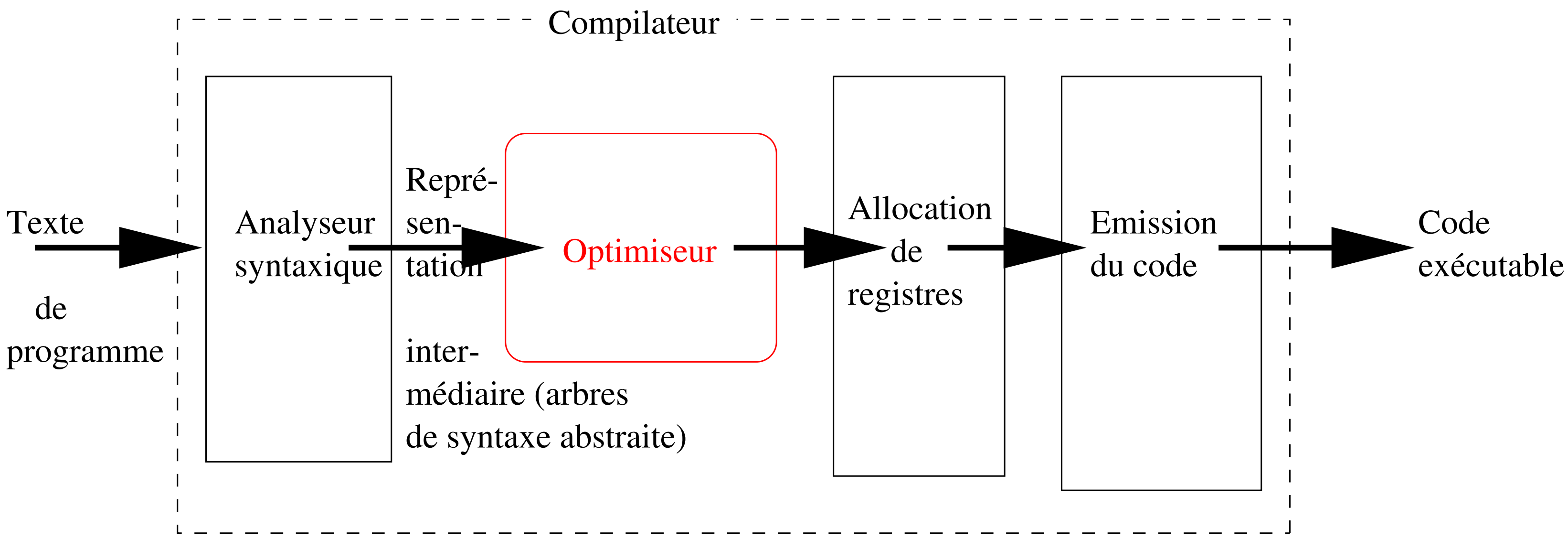
Jean Goubault-Larrecq

G.I.E. Dyade & INRIA

Jean.Goubault@dyade.fr

http://www.dyade.fr/fr/actions/vip/jglcours.html

Optimisation de programmes



Exemple : l’élimination de sous-expressions communes

```
int f (int i)
{
  int j = 2*i;
  int k = 2*i+1;
  int l = j*k;
  return l;
}
```

$\rightarrow$

```
int f (int i)
{
  int j = 2*i;
  int k = j+1;
  int l = j*k;
  return l;
}
```

Exemple : sortie d'invariants de boucles

```
void g(int n,int a[],
       int x) {
  int i;
  for (i=0; i<n; i++) {
    a[i] += x*x;
  }
}
```

$\rightarrow$

```
void g(int n,int a[],
       int x) {
  int i;
  int xx = x*x;
  for (i=0; i<n; i++) {
    a[i] += xx;
  }
}
```

Exemple : élimination de code mort (=jamais exécuté)

```
void h(int n,int a[],
       int x) {
  int i;
  for (i=0; i<n; i++) {
    a[i] += x;
  }
  if (i<n)
    printf ("Salut!");
}
```

$\rightarrow$

```
void h(int n,int a[],
       int x) {
  int i;
  for (i=0; i<n; i++) {
    a[i] += x;
  }
  (i<n);
}
```

Ex: Pourquoi `(i<n)` n'est-il pas mort ?

Exemple : élimination d'expressions inutiles

```
void h(int n,int a[],
       int x) {
  int i;
  for (i=0; i<n; i++) {
    a[i] += x;
  }
  (i<n);
}
```

$\rightarrow$

```
void h(int n,int a[],
       int x) {
  int i;
  for (i=0; i<n; i++) {
    a[i] += x;
  }
}
```

Ex: Remplacer `(i<n)` par `f(3)` : peut-on encore l'éliminer ?

Exemple : propagation de constantes

```
void e(int n,int a[]) {               void e(int n,int a[]) {
  int i;                                int i;
  int x = 4;                            int x = 4;
  for (i=0; i<n; i++) { →               for (i=0; i<n; i++) {
    a[i] += x*x+1;                        a[i] += 17;
  }                                     }
}                                     }
```

Ex: Pourrait-on supprimer le `int x = 4;` ?

Optimisation, conclusion

- Transformations de code justifiées par des propriétés sémantiques du programme (par ex., le fait que `x*x+1` vaille toujours `17` dans la dernière boucle);
- ces propriétés doivent être trouvées statiquement, c'est-à-dire sans avoir besoin d'exécuter le programme;

Détermination statique d'erreurs à l'exécution

```
int f (int i)
{
  int j;
  int k = i+j;
  j=3;
  return k;
}
```

Si on applique `gcc -c -O -Wall` dessus, on obtient :

```
opt2.c: In function `f':
opt2.c:4: warning: `j' might be used uninitialized
          in this function
```

Autres applications

- Production de jeux de test : génération de conditions nécessaires pour passer en chaque point du programme;
- Optimisation pour langages de haut niveau (Prolog, Haskell, ML, Java) : analyses de mode, de déterminisme, de nécessité, boxings/unboxings, etc.
- Gestion de ressources : parallélisation et vectorisation automatique;
- Debugging automatique : Ariane 5 et étreintes fatales; slicing;

et ...

Sûreté, sécurité

- Typage : garantir qu'il n'y aura pas d'erreur de type à l'exécution (ex.: on ne calcule pas la longueur de la chaîne `s` lorsque `s` est un entier);
- Pas d'erreur à l'exécution, plus généralement : pas de débordement arithmétique, pas de débordement de tableau, pas de débordement de pile (cf. Java bytecode verification), pas de déréférencement de pointeur `null`, absence de deadlocks, etc.
- Sécurité : pas d'interférence entre applications tournant sur une même machine, pas de flux d'information non souhaité (modèle de Bell-La Padula ou de la muraille de Chine).

Que peut-on faire ?

- Analyse de programmes dépendant de valeurs inconnues des données : le test (qui ne testera jamais qu'un nombre fini de valeurs possibles) ne suffit pas. On veut donc analyser les programmes sans les exécuter : l'analyse est statique.
- Demander les informations au programmeur ? Impossible à maintenir, impossible de faire confiance au jugement du programmeur. Il est nécessaire d'effectuer une analyse automatique.
- Mais une analyse exacte est impossible :

  Théorème (Rice) : Toute propriété extensionnelle non triviale portant sur des programmes est indécidable.

  $\Rightarrow$ nécessité d'une analyse approchée.

Le théorème de Rice

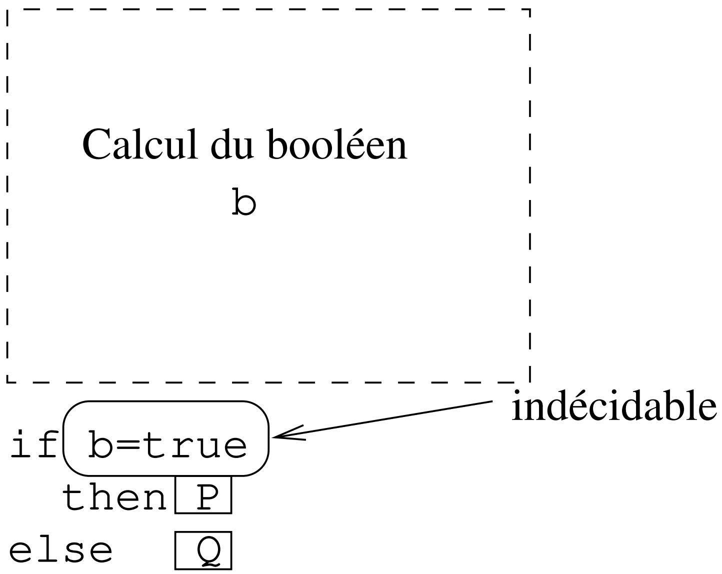
Toute propriété :

- extensionnelle — qui ne dépend que de la *sémantique* du programme, pas de sa syntaxe,
- et non triviale — ni toujours vraie ni toujours fausse,

est :

- indécidable — tout algorithme qui décide si cette propriété est vraie ou fausse boucle ou se trompe sur au moins une infinité de programmes.

Par exemple, “y a-t-il des exécutions rendant `b=true` ?” est indécidable :



Ex: en déduire que “`P` est-il mort ?” est indécidable.

Ex: montrer que “`Q` est-il mort ?” est indécidable.

Ex: montrer que “la valeur de `x` est-elle constante ?” est indécidable.

Ex: quelles sont les autres propriétés indécidables parmi celles mentionnées dans l’introduction ?

Comment l'interprétation abstraite marche-t-elle ?

- En première approche, l'interprétation abstraite est un mode de calcul non standard, où l'on remplace les vraies valeurs (les valeurs concrètes) par des symboles (les valeurs abstraites).
- À cause des problèmes d'indécidabilité, il est nécessaire de forcer parfois une perte d'information (approximation).

Un exemple d'approximation : la règle des signes

- Calcul standard (concret) : si $x = 3$ et $y = -4$, alors $x * x + y * y = 25 \geq 0$;
- Calcul non standard (abstrait) : si $x = +$ et $y = -$, alors $x * x + y * y = +$, où $+$ signifie "est positif ou nul", $-$ signifie "est négatif ou nul", et on a les tables :

$\_ + \_$	$+$	$-$
$+$	$+$	$\top$
$-$	$\top$	$-$

$\_ * \_$	$+$	$-$
$+$	$+$	$-$
$-$	$-$	$+$

et où $\top$ signifie "on ne sait pas".

Approximation

En fait, à cause de $\top$, on est obligé de compléter les tables :

$\_ + \_$	$+$	$-$	$\top$
$+$	$+$	$\top$	$\top$
$-$	$\top$	$-$	$\top$
$\top$	$\top$	$\top$	$\top$

$\_ * \_$	$+$	$-$	$\top$
$+$	$+$	$-$	$\top$
$-$	$-$	$+$	$\top$
$\top$	$\top$	$\top$	$\top$

- L'approximation est correcte : si $e = +$ en abstrait, alors $e \geq 0$ en concret; si $e = -$ en abstrait, alors $e \leq 0$ en concret.
- L'approximation perd de l'information : on ne peut pas déduire que $x = +$ et $y = -$ implique $x * x + 2 * x * y + y * y \geq 0$; ni que $x * x \geq 0$ lorsque le signe de x est inconnu.

Analyse de programmes : l'interprétation concrète

… ou calcul, ou sémantique (concrète).

Nous allons traiter l'exemple d'un langage jouet, une sorte de mini-Pascal idéalisé.

On part d'une syntaxe :

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

$$
\begin{array}{rcll}
e, \ldots \in \mathcal{E}xpr & \hat{=} & x \mid a & \text{expressions} \\
& \mid & e \,\texttt{+}\, e \mid \texttt{-}e \mid e * e \mid e \,/\, e \mid e \,\texttt{=}\, e \mid \texttt{not}\, e & \\
c, \ldots \in \mathcal{C}om & \hat{=} & x := e & \text{commandes} \\
& \mid & \texttt{input}\, x & \\
& \mid & \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2 & \\
& \mid & \texttt{while}\, e\, \{c\} & \\
& \mid & c_1 ; c_2 & \\
& \mid & \texttt{skip} &
\end{array}
$$

Syntaxe abstraite, en Caml (exemple)

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

Sémantique concrète (premier essai)

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

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

$$
\begin{array}{rcl}
\mathcal{C}[\![\_]\!]\_ & : & \mathcal{C}om \times \mathcal{E}nv \to \mathcal{E}nv \\
\mathcal{C}[\![x := e]\!]\rho & \hat{=} & \rho[x \leftarrow \mathcal{E}[\![e]\!]\rho] \\
\mathcal{C}[\![\mathtt{input}\ x]\!]\rho & \hat{=} & \rho[x \leftarrow ?] \\
\mathcal{C}[\![\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2]\!]\rho & \hat{=} & \begin{cases} \mathcal{C}[\![c_1]\!]\rho & (\mathcal{E}[\![e]\!]\rho = \mathtt{true}) \\ \mathcal{C}[\![c_2]\!]\rho & (\text{sinon}) \end{cases} \\
\mathcal{C}[\![\mathtt{while}\ e\ \{c\}]\!]\rho & \hat{=} & \begin{cases} \mathcal{C}[\![c; \mathtt{while}\ e\ \{c\}]\!]\rho & (\mathcal{E}[\![e]\!]\rho = \mathtt{true}) \\ \rho & (\text{sinon}) \end{cases} \\
\mathcal{C}[\![c_1 ; c_2]\!]\rho & \hat{=} & \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\rho) \\
\mathcal{C}[\![\mathtt{skip}]\!]\rho & \hat{=} & \rho
\end{array}
$$

où $\rho[x \leftarrow v]$ est l'environnement ρ' tel que $\rho'(y) \hat{=} \begin{cases} v & \text{si } x = y \\ \rho(y) & \text{sinon} \end{cases}$.

Seul petit défaut (!) : la clause pour `while` n'a aucun sens (pourquoi ?)

Sémantique concrète, en Caml (exemple)

```
exception Wrong;;    (* erreur de type *)
exception Err;;      (* autre erreur *)
type valeur = cst;; (* valeurs *)
type env = var -> valeur;;
let rec expr_eval rho e = match e with
          V x -> rho x
        | C c -> c
        | PLUS (e1, e2) ->
          (match (expr_eval rho e1, expr_eval rho e2) with
            (Z v1, Z v2) -> Z (v1 + v2) (* addition *)
          | (S v1, S v2) -> S (v1 ^ v2) (* concat. *)
          | _ -> raise Wrong)
        ... ;;
```

```
let rec com_eval rho c = match c with
          SET (x, e) ->
          (let v = expr_eval rho e
           in fun y -> if y=x then v else rho y)
        | INPUT x -> ...
        | IF (e, c1, c2) ->
          (match expr_eval rho e with
             B true  -> com_eval rho c1
           | B false -> com_eval rho c2
           | _ -> raise Wrong)
        | WHILE (e, c) ->
          (match expr_eval rho e with
             B true -> com_eval rho (SEQ (c, WHILE (e, c)))
           | B false -> rho
           | _ -> raise Wrong)
        | SEQ (c1, c2) -> com_eval (com_eval rho c1) c2
        | SKIP -> rho;;
```