Compilation

CM8 - Analyse de flot de données

ISTIC, Université de Rennes 1
Sebastien.Ferre@irisa.fr

COMP, M1 info

Plan

Plan

Rappels

Optimisation

→ conditions d'application

→ propriétés définies formellement en terme

- de graphe de flot de contrôle
- de flot de données (activité, visibilité)

Ces propriétés sont souvent indécidables

⇒ analyses approchées *approximation dans le bon sens !*

Automatisation de ces analyses

Analyses de complexité croissante :

1. analyses locales aux blocs de base
   pure séquences, pas de branchement
2. analyses globales avec flot de contrôle simple
   séquences, conditionnelles, itérations
   où l'on va retrouver nos grammaires attribuées !
3. analyses globales avec flot de contrôle quelconque
   `goto`, `call`

Plan

Analyse locale (sur blocs de base)

Illustration avec les variables actives

- motivation : supprimer du code mort
  - recherche de définitions sans utilisation
  - recherche de définitions non-active
- on va sur-approximer l'activité

Analyse utile et efficace

- si les variables actives en sortie de bloc sont connues
- cela mérite une analyse globale (voir plus loin)
- mais est approximable
  - les variables du programme source sont supposées active (pessimisme)
  - les variables temporaires sont inactives
    - car pas de branchement entre définition et utilisation
    - car utilisation locale à chaque construction (voir génération de code)

Analyse locale (sur blocs de base)

Illustration avec les variables actives

- motivation : supprimer du code mort
  - recherche de définitions sans utilisation
  - recherche de définitions non-active
- on va sur-approximer l'activité

Analyse utile et efficace

- si les variables actives en sortie de bloc sont connues
- cela mérite une analyse globale (voir plus loin)
- mais est approximable
  - les variables du programme source sont supposées active (pessimisme)
  - les variables temporaires sont inactives
    - car pas de branchement entre définition et utilisation
    - car utilisation locale à chaque construction (voir génération de code)

Analyse locale des variables actives

Principe :

- on part de la fin du bloc et on remonte dans le code (séquence)
- on maintient un sur-ensemble de variables actives A
  - chaque utilisation $\ldots a \ldots$ rend a active
  - chaque définition $a = \ldots$ rend a inactive
- initialement, A contient toutes les variables source

.....

Exemple

Analyse d'activité pour un programme factorielle

Réutilisation de variables temporaires

Remarque

La connaissance des variables actives permet aussi de réutiliser des variables temporaires de façon sûre.

Schéma : . . . . . .

Exemple

Réutilisation de temporaires dans le programme factorielle

Plan

Analyse globale

Illustration avec définitions visibles

- analyse locale : un bloc à la fois
- analyse globale : tous les blocs à la fois

Rappel

Une définition $(d)\ a = \ldots$ est visible en un point de programme p

- s'il existe un chemin partant du point d'entrée du programme,
- menant à p,
- et passant par d
- sans redéfinition de a entre d et p.

Applications de la visibilité

2 applications de la visibilité :

a) propagation de constantes/copies (sur-approximation)

. . . . . .

b) détection de variables *possiblement* non-initialisées (sous-approximation)

. . . . . .

Dans la suite : sur-approximation

Applications de la visibilité

2 applications de la visibilité :

a) propagation de constantes/copies (sur-approximation)

. . . . . .

b) détection de variables *possiblement* non-initialisées (sous-approximation)

. . . . . .

Dans la suite : sur-approximation

Instruction clé : l'affectation

schéma : . . . . . .

- Visible-après(d) = Visible-avant(d) \ Invisible-à-cause-de(d) $\cup$ Visible-à-cause-de(d)
- Visible-après($d : a = \ldots$) = Visible-avant(d) \ Définitions(a) $\cup \{d\}$
  - où Définitions(a) est l'ensemble des définitions (affectations) de a dans le programme

Instruction clé : l'affectation

schéma : . . . . . .

- Visible-après(d) = Visible-avant(d) $\setminus$ Invisible-à-cause-de(d) $\cup$ Visible-à-cause-de(d)
- Visible-après($d : a = \ldots$) = Visible-avant(d) $\setminus$ Définitions(a) $\cup \{d\}$
  - où Définitions(a) est l'ensemble des définitions (affectations) de a dans le programme

Généralisation à d'autres propriétés

Abstraction :

- Visible-après : propriété à la sortie de l'instruction $\rightarrow$ *Ex*
- Visible-avant : propriété à l'entrée de l'instruction $\rightarrow$ *In*
- Visible-à-cause-de $\rightarrow$ *Prod*
- Invisible-à-cause-de $\rightarrow$ *Suppr*

Pour toute instruction S :

$$Ex(S) = In(S) \setminus Suppr(S) \cup Prod(S)$$

Instanciation à la visibilité

- affectation : $S : (d)a = \ldots$
  - $Prod(S) = \{d\}$
  - $Suppr(S) = Definitions(a)$
- quid des autres instructions
  - séquences
  - branchements : conditionnelles et itérations

Flot de contrôle simple

Analyse sur du code 3-adresses structuré

- langage intermédiaire entre langage source et code 3-adresses
- on ne linéarise pas les structures de contrôle (séquences, conditionnelles et itérations)

$$
\begin{array}{lcl}
S & \rightarrow & \mathit{ident} = \mathit{ident}_1 \ \mathit{op} \ \mathit{ident}_2 \\
 & & \text{définition de } \mathit{ident}\text{, utilisation de } \mathit{ident}_1 \text{ et } \mathit{ident}_2 \\
 & | & S_1 \ ; \ S_2 \\
 & | & \underline{\mathit{if}} \ B \ \underline{\mathit{then}} \ S_1 \ \underline{\mathit{else}} \ S_2 \ \underline{\mathit{end}} \\
 & | & \underline{\mathit{do}} \ S_1 \ \underline{\mathit{while}} \ B \ \underline{\mathit{done}} \\
B & \rightarrow & \mathit{ident} \quad \text{utilisation de } \mathit{ident}
\end{array}
$$

Rappel : application à l'analyse de visibilité avec sur-approximation

Séquences

$S \rightarrow S_1 \ ; \ S_2 \ \ldots\ldots$

Conditionnelles

$S \rightarrow \underline{if}\ B\ \underline{then}\ S_1\ \underline{else}\ S_2 \ldots\ldots$

Impact du sens de l'approximation

- dans les conditionnelles, on fait une approximation à défaut de savoir quelle branche est exécutée
  - les équations pour l'affectation et la séquence étaient précises
- en cas de sous-approximation, on inverse les opérations $\cap$ et $\cup$
  - $Ex(S) = Ex(S_1) \cap Ex(S_2)$
  - $Prod(S) = Prod(S_1) \cap Prod(S_2)$
  - $Suppr(S) = Suppr(S_1) \cup Suppr(S_2)$

Itérations

$S \rightarrow \underline{do}\ S_1\ \underline{while}\ B\ \underline{done}\ \ldots\ldots$

On aboutit à une circularité dans les équations !

Itérations : suppression de la circularité

. . . . . .

Grammaires attribuées pour l'analyse de flots de données

- Pas de circularité dans les équations reliant $Ex(S), In(S), Prod(S), Suppr(S)$
  - ni dans séquences, ni dans conditionnelles
- On peut donc calculer les ensembles *Ex*, *In*, *Prod*, *Suppr* à l'aide d'une grammaire attribuée
  - grammaire : celle du code 3-adresses structuré
  - attribut hérité : *In*
  - attributs synthétisés : *Prod*, *Suppr*, *Ex*
  - équations : voir ci-dessus

Grammaire attribuée pour l'analyse de flot de données

Illustration avec définitions visibles

$$
\begin{array}{lll}
S & \rightarrow & \mathit{ident} = \mathit{ident}_1 \ \mathit{op} \ \mathit{ident}_2 \\
 & & \left\{ \begin{array}{lcl}
S.\mathit{Prod} & := & \text{cette definition} \\
S.\mathit{Suppr} & := & \text{toutes les definitions de } \mathit{ident} \\
S.\mathit{Ex} & := & S.\mathit{In} \setminus S.\mathit{Suppr} \cup S.\mathit{Prod}
\end{array} \right. \\
 & | & S_1 \ ; \ S_2 \\
 & & \left\{ \begin{array}{lcl}
S.\mathit{Prod} & := & S_1.\mathit{Prod} \setminus S_2.\mathit{Suppr} \cup S_2.\mathit{Prod} \\
S.\mathit{Suppr} & := & S_1.\mathit{Suppr} \setminus S_2.\mathit{Prod} \cup S_2.\mathit{Suppr} \\
S_1.\mathit{In} & := & S.\mathit{In} \\
S_2.\mathit{In} & := & S_1.\mathit{Ex} \\
S.\mathit{Ex} & := & S_2.\mathit{Ex}
\end{array} \right.
\end{array}
$$

sous-approximation / sur-approximation

$$
\begin{array}{lll}
S & \rightarrow & \underline{\mathit{if}}\ B\ \underline{\mathit{then}}\ S_1\ \underline{\mathit{else}}\ S_2 \\
 & & \left\{ \begin{array}{lcl}
S.\mathit{Prod} & := & S_1.\mathit{Prod} \cap / \cup\ S_2.\mathit{Prod} \\
S.\mathit{Suppr} & := & S_1.\mathit{Suppr} \cup / \cap\ S_2.\mathit{Suppr} \\
S_1.\mathit{In} & := & S.\mathit{In} \\
S_2.\mathit{In} & := & S.\mathit{In} \\
S.\mathit{Ex} & := & S_1.\mathit{Ex} \cap / \cup\ S_2.\mathit{Ex}
\end{array} \right. \\
 & | & \underline{\mathit{do}}\ S_1\ \underline{\mathit{while}}\ B\ \underline{\mathit{done}} \\
 & & \left\{ \begin{array}{lcl}
S.\mathit{Prod} & := & S_1.\mathit{Prod} \\
S.\mathit{Suppr} & := & S_1.\mathit{Suppr} \\
S_1.\mathit{In} & := & S.\mathit{In} \cap / \cup\ S_1.\mathit{Ex} \\
 & = & S.\mathit{In} \setminus (S_1.\mathit{Suppr} \setminus S_1.\mathit{Prod}) \\
 & / & S.\mathit{In} \cup S_1.\mathit{Prod} \\
S.\mathit{Ex} & := & S_1.\mathit{Ex}
\end{array} \right.
\end{array}
$$

Exemple

Analyse de visibilité pour le programme factorielle

Flot de contrôle quelconque

C'est l'ajout de branchements arbitraires : `goto`, `call`

- complexifie le graphe de flot de contrôle
- crée des dépendances circulaires entre les attributs
- ne permet plus de faire le calcul par grammaire attribuée

Peut-on l'éviter ?

- oui pour le `goto` (conditionnelles et itérations)
- non pour les appels de fonctions (`call`) qui se comportent comme des `goto` sophistiqués

Flot de contrôle quelconque

C'est l'ajout de branchements arbitraires : `goto`, `call`

- complexifie le graphe de flot de contrôle
- crée des dépendances circulaires entre les attributs
- ne permet plus de faire le calcul par grammaire attribuée

Peut-on l'éviter ?

- oui pour le `goto` (conditionnelles et itérations)
- non pour les appels de fonctions (`call`) qui se comportent comme des `goto` sophistiqués

Principe

L'analyse d'un flot de contrôle quelconque suit le principe suivant :

1. On calcule les *Prod* et *Suppr* pour chaque bloc
   - bloc = séquence sans branchement
   - comme précédemment, avec grammaire attribuée, de façon synthétisé
2. On initialise $B.In$ et $B.Ex$ pour chaque bloc B
3. Pour chaque bloc B, on évalue les équations
   - $B.Ex := B.In \setminus B.Suppr \cup B.Prod$
   - $B.In := \bigcap / \bigcup_{P \in Pred(B)} P.Ex$
     - sous-approximation / sur-approximation
     - $Pred(B)$: blocs prédécesseurs de B dans le graphe de flot de contrôle
     - graphe avec cycle $\Rightarrow$ circularité dans les équations
4. Répéter (3) jusqu'à stabilisation

Conditions de convergence

- les valeurs initiales de *Ex* et *In* sont
  - sous-approximation : les plus grandes
    ex : toutes les définitions du programme
  - sur-approximation : les plus petites ($\emptyset$)
- l'évaluation des équations ne peut que diminuer/augmenter les valeurs (monotonie)
- tout ensemble de valeurs à une borne inférieure/supérieure
  ex : intersection / union

Analyse arrière

Exemple : analyse de variables actives

- rappel : dans un bloc, on procède en remontant le code, de la fin du bloc vers le début
- on parle d'analyse en arrière
- l'analyse de visibilité était en avant

On inverse les rôles de *In* et *Ex* dans les équations :

- $B.In := B.Ex \setminus B.Suppr \cup B.Prod$
- $B.Ex := \bigcap / \bigcup_{S \in Succ(B)} S.In$

Analyse arrière

Exemple : analyse de variables actives

- rappel : dans un bloc, on procède en remontant le code, de la fin du bloc vers le début
- on parle d'analyse en arrière
- l'analyse de visibilité était en avant

On inverse les rôles de *In* et *Ex* dans les équations :

- $B.In := B.Ex \setminus B.Suppr \cup B.Prod$
- $B.Ex := \bigcap / \bigcup_{S \in Succ(B)} S.In$

Example

Analyse de variables active du programme factorielle

- avec flot de contrôle quelconque
- avec analyse arrière
- avec sur-approximation

. . . . . .