Compilation

CM7 - Optimisations

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

COMP, M1 info

Plan

Plan

Motivation

On a vu que la génération de code intermédiaire produisait du code inefficace :

- parce que la génération est locale à chaque construction

$\Rightarrow$ on a besoin d'une vision globale du code pour l'optimiser

- ex. code généré : `t1 = t2; ...; t2 = t1`
- la 2ème instruction `t2 = t1` peut être supprimée
  1. si pas de saut arrivant entre les deux instructions
  2. si ni `t1` ni `t2` modifié entre les deux instructions

Optimisation

Definition

Appliquer une optimisation, c'est appliquer une transformation sur un morceau du code lorsque celui-ci satisfait certaines conditions garantissant la préservation de la sémantique du programme.

Problème d'optimisation

Sur l'exemple précédent :

- le problème est de savoir si on peut supprimer la 2ème instruction
- une instance du problème est un couple d'instruction (`t1 = t2`, `t2 = t1`)
- une instance est positive
  c-à-d. on peut appliquer l'optimisation
  si on les conditions d'application sont vérifiées
  et est négative sinon

Remarque

Un problème d'optimisation est décidable s'il existe un algorithme qui sait dire pour toute instance si elle est positive ou négative

Nécessité de l'approximation

Le plus souvent, les problèmes d'optimisation sont indécidables

- car ils portent sur la sémantique des programmes [théorème de Rice]

→ on va les approximer

- le terme "optimisation" est donc impropre
- on ne trouve pas la "meilleure" solution

On devrait plutôt parler d'"amélioration de la performance"

- en temps : moins d'instructions, instructions moins coûteuses
- en mémoire : moins d'emplacement mémoire, meilleure localité

Nécessité de l'approximation

Le plus souvent, les problèmes d'optimisation sont indécidables

- car ils portent sur la sémantique des programmes [théorème de Rice]

→ on va les approximer

- le terme "optimisation" est donc impropre
- on ne trouve pas la "meilleure" solution

On devrait plutôt parler d'"amélioration de la performance"

- en temps : moins d'instructions, instructions moins coûteuses
- en mémoire : moins d'emplacement mémoire, meilleure localité

Sens de l'approximation

L'approximation peut se faire dans deux sens opposés :

- sous-approximation : $Pos_{approx} \subseteq Pos$
- sur-approximation : $Pos_{approx} \supseteq Pos$
- diagramme de Venn : . . . . . .

Question : laquelle est préférable dans l'exemple précédent ?

Sens de l'approximation

Il convient d'être pessimiste, prudent !

- *dans le doute, s'abstenir (d'optimiser)*
- indécision $\Rightarrow$ négatif
- pour éviter d'appliquer une optimisation en dehors de ses conditions d'application
- on préfère manquer une opportunité d'optimisation que de ne pas préserver la sémantique !
- diagrammes de Venn : . . . . . .

Alors, sous-approximation ou sur-approximation ?

- cela dépend de comment le problème et les conditions sont formulées
- exemple :
  - sous-approximation de *pas de saut entre les 2 instructions*
  - = sur-approximation de *saut entre les 2 instructions*

Sens de l'approximation

Il convient d'être pessimiste, prudent !

- *dans le doute, s'abstenir (d'optimiser)*
- indécision $\Rightarrow$ négatif
- pour éviter d'appliquer une optimisation en dehors de ses conditions d'application
- on préfère manquer une opportunité d'optimisation que de ne pas préserver la sémantique !
- diagrammes de Venn : . . . . . .

Alors, sous-approximation ou sur-approximation ?

- cela dépend de comment le problème et les conditions sont formulées
- exemple :
  - sous-approximation de *pas de saut entre les 2 instructions*
  - = sur-approximation de *saut entre les 2 instructions*

Plan

Simplifications algébriques

D'après les propriétés des opérateurs arithmétiques et logiques :

- éléments neutres
  - $0 + x \rightarrow x$
  - $1 * x \rightarrow x$
  - *false or* $x \rightarrow x$
  - *true and* $x \rightarrow x$
- éléments absorbants
  - $0 * x \rightarrow 0$
  - *false and* $x \rightarrow$ *false*
  - *true or* $x \rightarrow$ *true*

Élimination des sous-expressions communes

Definition

Appliquer la transformation

$$t_i = a \; op \; b; \ldots_1 ; t_j = a \; op \; b \longrightarrow t_i = a \; op \; b; \ldots_1 ; t_j = t_i$$

si

1. pas de saut entre (les 2 instructions)
2. aucun de t_i, a, b ne peut être modifié entre

Exemple : . . . . . .

Propagation de copies

ou plutôt : propagation de l'original

Definition

Appliquer la transformation

$$x = y; \ldots_1 ; \ldots_2 = \ldots_3 x \ldots_4 \longrightarrow x = y; \ldots_1 ; \ldots_2 = \ldots_3 y \ldots_4$$

si

1. pas de saut entre
2. aucun de x, y ne peut être modifié entre

Exemple : . . . . . .

Élimination de code mort

Definition

Appliquer la transformation

$$\ldots_1 ; x = e; \ldots_2 \longrightarrow \ldots_1 ; \ldots_2$$

si

1. la variable x n'est pas lue avant d'être affectée de nouveau

nécessite de connaitre tous les chemins d'éxécution

Exemple : . . . . . .

Interdépendances de ces optimisations

Elles s'alimentent les unes les autres :

1. élimination des sous-expressions communes
   - supprime des opérations *a op b*
   - produit des copies $x = y$
2. propagation de copies
   - ne fait rien gagner en soi
   - mais produit du code mort
3. élimination du code mort
   - supprime des instructions (affectations)

Remarque

Ces optimisations suffisent à supprimer beaucoup de variables intermédiaires produites par la génération de code

Interdépendances de ces optimisations

Elles s'alimentent les unes les autres :

1. élimination des sous-expressions communes
   - supprime des opérations *a op b*
   - produit des copies $x = y$
2. propagation de copies
   - ne fait rien gagner en soi
   - mais produit du code mort
3. élimination du code mort
   - supprime des instructions (affectations)

Remarque

Ces optimisations suffisent à supprimer beaucoup de variables intermédiaires produites par la génération de code

Optimisation des boucles

Idée générale : diminuer le coût des boucles les plus imbriquées $\Rightarrow$ c-à-d. celles exécutées le plus souvent

Definition

Appliquer la transformation

$$\textit{while}\ \ldots_0\ \textit{do}\ \ldots_1 ; x = e; \ldots_2\ \textit{done} \longrightarrow$$

$$x = e;\ \textit{while}\ \ldots_0\ \textit{do}\ \ldots_1 ; \ldots_2\ \textit{done}$$

si d'une itération à l'autre

1. aucune variable de x et e ne peut être modifiée

Optimisation des boucles

Exemple (sur du code 3-adresses structuré) : . . . . . .

Plan

Quelques propriétés utiles

Les conditions d'application des optimisations s'expriment en termes de propriétés du programme

- propriété du point d'entrée unique
  - ex : pas de saut entre...
- propriété d'être actif
  - ex : variable utilisée...
- propriété d'être visible

Remarque

Ces propriétés apportent beaucoup d'information sur le programme «en actes», c-à-d. sa sémantique, son comportement à l'exécution.

Propriété du point d'entrée unique

- condition "pas de saut entre deux instructions (i) et (j)"
  - peut-on exécuter (i) sans exécuter (j)
  - peut-on exécuter (j) sans exécuter (i)
- on veut connaître les chemins d'exécution possibles
  - le graphe de flot de contrôle est une structure de données représentant un programme qui répond précisément à cette question

Graphe de flot de contrôle

Definition

Le graphe de flot de contrôle d'un programme (code 3-adresses) est un graphe dont

- les noeuds sont les instructions
- les arcs sont
  - les branchements inconditionnels (goto)
  - les branchements conditionnels (ifz, ifnz)
  - les passages en séquences (branchements par defaut)

On suppose :

- 1 point d'entrée (sans prédécesseur) : début du programme
- 1 point de sortie (sans successeur) : fin du programme

Blocs de base

On peut simplifier le graphe de flot de contrôle en remplaçant les instructions par des blocs de base

Definition

Un bloc de base est un chemin (du graphe de flot de contrôle) de longueur maximale ayant au plus

- un noeud avec 0 ou plusieurs prédécesseurs (point d'entrée du bloc)
- un noeud avec 0 ou plusieurs successeurs (point de sortie du bloc)

Schéma : . . . . . .

Calcul des blocs de base

- algorithme quasi-linéaire
  - 1 parcours du code
  - segmentation de la séquence d'instructions
  - débuts de blocs : début de programme, `label`
  - fins de blocs : fin de programme, `goto`, `ifz`, `ifnz`
  - ……

Exemple de graphe de flot de contrôle

code, blocs de base, graphe, chemins possibles

. . . . . .

Définitions et utilisations de variables dans les instructions

- Notations
  - $a = \ldots$: instruction définissant a
  - $\ldots a \ldots$: instruction utilisant a
- Exemples d'instructions
  1. $a = b \; op \; c$: utilise b et c et définit a
  2. *ifz a goto L* : utilise a (et ne définit rien)
  3. $a = call \; L$: définit a (et n'utilise rien)

Utilisations entre instructions, activité, durée de vie

Definition

Une instruction (i) $a = \ldots$ est utilisée par une instruction (j) $\ldots a \ldots$ s'il existe un chemin

- partant du point d'entrée du programme — éviter code mort
- passant par (i) — définition
- puis passant par (j) — utilisation
- sans re-définir a entre (i) et (j)

- (i) est dite active entre (i) et (j)
  - active = a été définie + sera utilisée avant d'être re-définie
- la durée de vie d'une définition est l'ensemble des points de contrôle (instructions) où elle est active

Exemple d'activité

Dans l'exemple précédent

- sortie bloc B1 : t_1 pas actif, i actif car utilisé dans B2
- durée de vie de t_1 : instruction 2

Application : élimination de code mort

La propriété d'être actif fournit le critère formel pour l'élimination de code mort

- condition : "la variable x n'est pas lue avant d'être affectée de nouveau"
- équivalent à "l'instruction à éliminer $x = e$ n'est active nulle part"
  - sous-approximer la non-activité
  - donc sur-approximer l'activité
- exemples
  - `a = 12; ifz x goto L1; b = a; label l1`
    `a = 12` active ?
  - `a = 12; ifz x goto L1; a = 3; b = a; label l1`
    `a = 12` active ?

Paires définition-utilisations

Definition

Les paires définition-utilisations (DU) d'un programme est l'association à chaque instruction définissant une variable de l'ensemble des instructions qui l'utilisent.

Exemple sur le graphe précédent : . . . . . .

Propriété d'être visible

Question

Devant une instruction (u) $\ldots a \ldots$, de quelles définitions peut venir la valeur de a ?

Il peut y en avoir plusieurs !

Definition

Une définition (d) $a = \ldots$ est visible en une instruction d'utilisation (u) $\ldots a \ldots$

- si (d) est active en (u)
- si (u) utilise (d)

C'est un autre point de vue sur la même information

Comparaison de l'activité et de la visibilité

actif	visible
centré définition	centré utilisation
paire (définition, utilisations)	paire (utilisation, définitions)
analyse en avant	analyse en arrière
.....	

Plan

Allocation de registres

- La génération de code alloue de nombreux emplacements mémoires
  - indifférenciés dans le code intermédiaire
  - différenciés dans le code cible
    - registres : accès rapide mais peu nombreux
    - mémoire : accès lent mais nombreux
- Les optimisations précédentes ont permis de réduire le nombre d'emplacements distincts
- On veut allouer un maximum de ces emplacements en registre

Un problème de coloriage de graphes

Le problème de l'allocation de registres peut être formalisé comme un problème de coloriage de graphe

- le graphe à colorier
  - noeuds : emplacements mémoires utilisés
  - arcs : interférences entre emplacements
    - x et y interfèrent si leurs durées de vie se chevauchent
    - c-à-d. si x et y sont actives en un point de programme
- le coloriage
  - consiste à attribuer une couleur à chaque noeud (emplacement)
  - tel que deux noeuds reliés par un arc (interférence) aient des couleurs différentes

Exemple de programme et graphe d'interférence

```
1   A = read();
2   B = read();
3   C = read();
4   A = A + B + C;
5   if(A < 10){
6       D = C + 8;
7       print(C);
    }else
8   if(A > 20){
9       E = 10;
10      D = E + A;
11      print(E);
    }else{
12      F = 12;
13      D = F + A;
14      print(F);
    }
15  print(D);
```

.....

Algorithme de coloriage

- problème NP-complet
  - pas d'algorithme polynomial
- algorithme avec approximation
  - solution pas forcément optimale
    on risque de consommer plus de registres que nécessaire
  - à base d'heuristique : on élimine successivement les noeuds les moins connectés, puis on les colorie dans l'ordre inverse d'élimination
  - en pratique : très bonne approximation
  - les noeuds qui ne peuvent pas être coloriés correspondent à des emplacements qui seront placés en mémoire

Coloriage de l'exemple

. . . . . .