Introduction aux optimisations

Martin Odersky

29 janvier 2007
version 1.3

Plan du cours

Optimisations simples

Ce cours présente quelques exemples d'optimisations simples qui pourraient facilement être ajoutées au compilateur Zwei.

- Le point commun de toutes ces optimisations est qu'elles peuvent s'effectuer directement sur l'arbre de syntaxe abstraite.
- Certaines optimisations plus avancées requièrent une autre représentation du code intermédiaire.

Réduction d'expressions constantes

Exemple

Pour l'expression `4 * 7`, on attend d'un bon compilateur qu'il émette l'instruction `ORIU R1 R0 28`.

Le calcul des expressions constantes (*constant folding*) se fait à la compilation.

On peut facilement ajouter cela dans une phase préalable de transformation de l'arbre, voir le faire directement à la construction de l'arbre ou durant l'analyse de types.

Réduction de force des opérateurs

Certaines opérateurs «coûteux» peuvent être remplacés par d'autres opérateurs plus simples si une des opérandes est une constante particulière :

- l'expression `x * 2^n` devient `x << n`,
- l'expression `x / 2^n` devient `x >> n`,
- l'expression `x % 2^n` devient `x & (2^n-1)`.

Les deux dernières sont seulement correctes pour des `x` positifs.

- Pour des `x` négatifs il faut que la division arrondisse vers le bas, et pas vers zéro ($-3/10$ doit valoir -1).
- C, Java et la plupart des processeurs font le contraire.

Élimination de sous-expressions communes

Exemple

Soit :

```
Int x = (a + b) / c;
Int y = (a + b) / d;
```

Le compilateur pourrait évaluer `(a + b)` une seule fois :

```
Int temp = (a + b);
Int x = temp / c;
Int y = temp / d;
```

On appelle cette optimisation élimination de sous-expressions communes (*common sub-expressions elimination*, *CSE*).

Pourquoi ne pas demander au programmeur d'écrire lui-même la version optimisée ?

- le premier programme peut être jugé plus clair,
- parfois les sous-expressions communes apparaissent dans du code que le programmeur n'a pas écrit (en Java, `a[i] = b[i]` contient implicitement deux sous-expressions de la forme `R = i * 4` pour le calcul de l'adresse des éléments du tableau).

L'élimination de sous-expressions communes ne constitue pas toujours une amélioration :

- Le stockage et le chargement des variables temporaires introduites ont un coût.
- Ce coût est négligeable si les variables temporaires ne sont jamais stockées en mémoire.
- Si l'élimination de sous-expressions communes introduit trop de variables temporaires stockées dans des registres, il est possible qu'il ne reste que peu de registres pour stocker d'autres données ; la pression sur les registres augmente.
- On n'effectue l'élimination de sous-expressions communes que si on a assez de registres à disposition.

Détection de sous-expressions communes

Exemple

```
Int x = (a + b) / c;
a = a + 1;
Int y = (a + b) / d;
```

`(a + b)` n'est pas une sous-expression commune car la variable `a` est modifiée dans le second énoncé.

Ce problème se pose dans tous les langages qui admettent les modifications de variables.

Pour détecter les sous-expressions communes, on utilise la numérotation des versions :

- lors de la production de code, on donne à chaque variable locale un numéro de version (dans le symbole, par exemple),
- chaque affectation à la variable incrémente sa version,
- on recherche ensuite les sous-expressions communes dans lesquelles les numéros de version des variables utilisées concordent.

Exemple

L'exemple précédent devient, une fois versionné :

```
x1 = (a1 + b1) / c1;
a2 = a1 + 1;
y1 = (a2 + b1) / d1;
```

La fausse sous-expression commune `(a + b)` a désormais disparu.

Pour trouver les expressions qui ont déjà été évaluées, on a en général recours à une table de hachage associant des arbres à des variables temporaires ;

en Scala, elle aurait le type `HashMap[Tree, Symbol]`.

Question : Que doit-on ajouter à la classe `Tree` pour l'utiliser dans une table de hachage ?

Optimisations avancées

Les optimisations précédentes ne sont que le début. Les compilateurs réels en effectuent bien d'autres, et on en découvre constamment de nouvelles.

Les principales sources d'optimisations sont :

- Limiter la «stupidité» en :
  - supprimant le code mort (*dead code elimination*),
  - supprimant les sauts vers d'autres sauts,
  - supprimant les chargements de variable suivis d'un stockage de la même variable.
- Améliorer l'utilisation de la mémoire en :
  - stockant les variables dans les registres,
  - améliorant la «localité» des accès aux variables, pour qu'elles restent dans l'antémémoire du processeur.

- Augmenter le parallélisme en :
  - ordonnant les instructions pour que plus d'instructions puissent être exécutées en parallèle et pour éviter les «bulles» dans le pipeline,
  - réécrivant les programmes pour diminuer le nombre de sauts (dérouler les boucles, *loop unroll*),
  - réécrivant les programmes pour utiliser les fils d'exécution multiples (*threads*).
- Éviter les calculs répétés en :
  - déplaçant les instructions d'un endroit où elles sont exécutées souvent vers un endroit où elles sont exécutées moins souvent (déplacer les instructions en dehors d'une boucle si leur valeur est invariante par rapport à la boucle).