

Compilation et langages : optimisations

Thibaut Balabonski

4 novembre 2016

Programmes optimisés

Objectif de l'optimisation : obtenir des programmes plus efficaces.

- Plus rapides
- Consommant moins de mémoire
- Plus compacts

L'optimisation des programmes peut être faite par le programmeur lui-même, notamment par le choix de bons algorithmes et de bonnes structures de données. Le compilateur peut ensuite agir à chacune de ses étapes, des phases intermédiaires à la production de code assembleur.

- Une optimisation lors d'une phase intermédiaire travaille sur une représentation intermédiaire (par exemple sur un AST) et est a priori générique.
- Une optimisation lors de la production de code assembleur est spécifique à une architecture cible.

Principe

Les phases d'optimisation peuvent être très coûteuses au moment de la compilation, pour n'apporter que des modifications mineures du programme. Mais :

- on compile une fois,
- on exécute le programme n fois,
- on exécute certaines instructions un grand nombre de fois à chaque exécution du programme.

Les petits gains sur des opérations répétées peuvent rapporter beaucoup à terme, et il peut être particulièrement intéressant d'optimiser les instructions les plus souvent répétées. En particulier, les boucles contiennent des instructions exécutées plusieurs fois, et cet effet est multiplié quand une boucle est à l'intérieur d'une autre boucle : les boucles les plus internes sont potentiellement des points clés dans le temps d'exécution d'un programme.

Que peut-on optimiser ?

Passons en revue un certain nombre d'exemples dans lesquels des optimisations sont possibles ou impossibles.

Simplification des expressions constantes (*constant folding*) Dans le fragment de code suivant

```
print_int (1 + 2)
```

le résultat de l'expression `1 + 2` peut être déterminé pendant la compilation plutôt que délégué au code assembleur produit. On peut donc produire du code correspondant à

```
print_int 3
```

Ce genre de simplification est impossible que lorsque l'un des opérandes n'est pas directement une valeur :

```
print_int (1 + x)
print_int (1 + f(x))
```

Utilisation d'identités arithmétiques Certaines expressions peuvent être simplifiées en utilisant des propriétés arithmétiques de base :

```
print_int (0 * (1 + 2))    ==>    print_int 0
print_int (0 * x)          ==>    print_int 0
while (true || x) { ... } ==>    while (true) { ... }
```

Cependant, ceci ne doit pas être fait si le calcul du deuxième opérande est susceptible de générer des effets de bord :

```
print_int (0 * f(1))      (* [f] peut avoir des effets sur la mémoire *)
print_int (0 * (2 / 0))   (* La division par [0] déclenche une exception *)
```

Propagation des constantes (*constant propagation*) Dans le fragment de code suivant

```
let x = 1
print_int x
```

la valeur de la variable `x` au niveau de l'instruction `print` peut être prédite. On peut donc directement produire du code correspondant à

```
let x = 1
print_int 1
```

et se passer de l'accès mémoire récupérant dynamiquement la valeur de `x`. Ceci devient impossible si la valeur de `x` est ambiguë ou s'il y a un risque qu'elle ait été modifiée :

```
if b then x := 1 else x := 2; print_int !x    (* [1] ou [2] ? *)
x := 1; f(2); print_int !x                    (* [f] a-t-elle modifié [x] ? *)
```

Élimination du code inutile (*dead code elimination*) Le calcul et l'affectation d'une valeur à une variable peuvent être évités si la variable n'est jamais utilisée. Notamment, si la variable `x` n'est utilisée nulle part ailleurs, le code précédent

```
let x = 1
print_int 1
```

peut être remplacé par

```
print_int 1
```

Comme précédemment, ceci sous-entend que le calcul de la valeur affectée à `x` ne génère pas d'effet de bord. On ne peut pas supprimer un calcul de la forme

```
let x = f(1)
```

Élimination des sous-expressions communes (*common subexpression elimination*) Le calcul d'une valeur qui a à *coup sûr* déjà été calculée peut être évité :

```
print_int (2 * 3)
print_int (1 + (2 * 3))
```

peut être transformé en

```
let z = 2 * 3
print_int z
print_int (1 + z)
```

et

```
if b
then x := 2 * 3
else x := 1 + (2 * 3)
print_int (2 * 3)
```

peut être transformé en

```
let z = 2 * 3
if b then x := z else x := 1 + z
print_int z
```

En revanche, dans le code suivant la valeur de `2 + !x` n'est pas réutilisable :

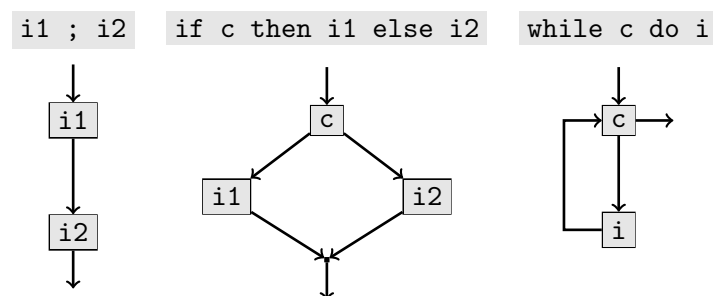
```
print_int (2 + !x)
x := 1
print_int (3 * (2 + !x))
```

Raisonner sur les exécutions possibles

La bonne application des transformations précédentes demande de savoir à différents points du programme quels sont les scénarios d'exécution possibles et leurs effets en termes de définition ou de modification de variables, ou de calcul d'expressions.

Le *graphe de flot de contrôle* permet de résumer les différentes exécutions possibles :

- Ses nœuds sont les instructions (ou blocs d'instructions) du programme.
- Ses arêtes représentent les enchaînements possibles entre deux blocs ou instructions.



Toute exécution du programme suit l'un des chemins du graphe de flot de contrôle. En revanche, certains chemins du graphe de flot de contrôle ne sont empruntés par aucune exécution réelle du programme : les chemins du graphe de flot de contrôle donnent une *surapproximation* des comportements possibles du programme.

Équations de flot de données

Pour résumer les comportements des exécutions du programme, à chaque nœud du graphe de flot de contrôle on peut ajouter des informations sur les actions passées ou à venir (par exemple modifications ou utilisations de variables). Plus précisément, un nœud du programme correspondant potentiellement à un changement d'état, on séparera les informations d'un nœud en des informations d'entrée et des informations de sortie.

Dans la suite nous allons voir trois exemples, qui donneront trois variantes d'une même technique.

Définitions accessibles Une instruction $\mathbf{x} := \mathbf{e}$ définit une nouvelle valeur pour une variable \mathbf{x} . En un point ultérieur du programme, cette définition est dite *accessible* si la valeur courante de \mathbf{x} peut être celle affectée par cette instruction.

Supposons qu'on veuille connaître pour chaque utilisation de variable les sources potentielles des valeurs lues. Nous allons chercher à déterminer l'ensemble des définitions accessibles à l'entrée ($In[n]$) et à la sortie ($Out[n]$) de chaque point n du programme. Les effets des différentes instructions sur ces ensembles sont les suivantes :

- En sortie d'un nœud n contenant une affectation $\mathbf{x} := \mathbf{e}$, la seule définition accessible pour \mathbf{x} est cette instruction (les éventuelles définitions de \mathbf{x} accessibles en entrée étant effacées par celle-ci). Les définitions accessibles des autres variables sont en revanche les mêmes en sortie qu'en entrée :

$$Out[n] = (In[n] \setminus \{\text{définitions de } \mathbf{x}\}) \cup \{n\}$$

- Pour un nœud n dont l'instruction ne modifie la valeur d'aucune variable, les définitions accessibles en entrée et en sortie sont les mêmes.

$$Out[n] = In[n]$$

- Lorsque deux nœuds p et n se succèdent, les définitions accessibles en entrée du deuxième sont égales aux définitions accessibles en sortie du premier.

$$In[n] = Out[p]$$

- Lorsque deux chemins du graphe se rejoignent, c'est-à-dire quand deux arêtes convergent vers un même nœud n depuis des nœuds p_1 et p_2 , alors les définitions accessibles en entrée de n sont l'union des définitions accessibles en sortie de chaque prédécesseur.

$$In[n] = Out[p_1] \cup Out[p_2]$$

On systématise l'écriture de ces équations en associant deux ensembles supplémentaires à chaque nœud n :

- $Kill[n]$ est l'ensemble des définitions supprimées par un nœud, c'est-à-dire l'ensemble de définitions de \mathbf{x} si le nœud contient une affectation $\mathbf{x} := \mathbf{e}$,
- $Gen[n]$ est l'ensemble des définitions produites par un nœud, c'est-à-dire le nœud lui-même s'il est une affectation $\mathbf{x} := \mathbf{e}$, et l'ensemble vide sinon.

La forme finale des *équations de flot de données* est alors :

$$\begin{aligned} Out[n] &= (In[n] \setminus Kill[n]) \cup Gen[n] \\ In[n] &= \bigcup_{p \in Pred[n]} Out[p] \end{aligned}$$

où $Pred[n]$ désigne l'ensemble des prédécesseurs du nœud n dans le graphe de flot de contrôle.

Expressions disponibles À un point de programme donné, on dit qu'une expression \mathbf{e} est *disponible* si elle a nécessairement été calculée dans tout chemin d'exécution menant à ce point.

Appuyons sur la principale différence avec l'exemple précédent :

- Une définition accessible est une définition présente dans *un* chemin du graphe.
- Une expression disponible est une expressions rencontrée dans *tous* les chemins du graphe.

Cette variation se traduit par un remplacement de l'union des chemins précédents par leur intersection, et les équations de flot de données pour les expressions disponibles ont donc la forme :

$$\begin{aligned} Out[n] &= (In[n] \setminus Kill[n]) \cup Gen[n] \\ In[n] &= \bigcap_{p \in Pred[n]} Out[p] \end{aligned}$$

Pour que ces équations soient complètes, il ne reste qu'à donner des définitions adéquates aux ensembles $Kill$ et Gen :

- $Kill[n]$ est l'ensemble vide, pour tout n .
- $Gen[n]$ est l'ensemble des sous-expressions présentes dans l'instruction du nœud n .

Variables vivantes À un point n donné d'un programme, une variable x est dite *vivante* si sa valeur actuelle est susceptible d'être utilisée dans le futur.

Cette information de vivacité présente une différence fondamentale avec les deux exemples précédents : c'est des instructions venant *après* le nœud n qu'on la déduit. Les équations de flot de données vont donc être orientées dans l'autre sens, de la sortie d'un nœud vers son entrée, et des successeurs vers les prédécesseurs. En revanche, il suffit qu'une variable soit utilisée dans *un* chemin du graphe pour qu'elle soit considérée vivante, la combinaison des informations de vivacité de plusieurs successeurs se fera donc par une union.

On obtient alors des équations de la forme

$$\begin{aligned} In[n] &= (Out[n] \setminus Kill[n]) \cup Gen[n] \\ Out[n] &= \bigcup_{s \in Succ[n]} In[p] \end{aligned}$$

Les ensembles *Kill* et *Gen* sont définis de la sorte :

- Une affectation à une variable x supprime cette variable : $Kill[n] = \{x\}$ si le nœud n est une instruction $x := e$, et $Kill[n]$ est l'ensemble vide sinon.
- $Gen[n]$ est l'ensemble des variables dont les valeurs sont lues par l'instruction du nœud n .

Résolution itérative des équations de flot de données

Si le graphe de flot de contrôle ne contient pas de cycle, c'est-à-dire si le programme que l'on cherche à analyser ne contient pas de boucles, alors la résolution des équations est simple : il suffit de parcourir l'ensemble des nœuds une fois (dans le bon ordre).

En revanche, dans le cas général le graphe de flot de contrôle contient des cycles qui induisent des dépendances cycliques entre les équations des différents nœuds. L'objectif est alors de chercher un *point fixe* de l'ensemble des équations.

Résolution pour les définitions accessibles On peut obtenir le *plus petit point fixe* de nos équations de flot de données avec la méthode suivante :

1. Pour tout nœud n , initialiser les ensembles $In[n]$ et $Out[n]$ à \emptyset .
2. Pour chaque nœud n (pris dans un ordre arbitraire) :
 - (a) Mettre à jour $In[n]$ suivant l'équation $In[n] = \bigcup_{p \in Pred[n]} Out[p]$ en utilisant les valeurs actuelles des $Out[p]$.
 - (b) Éventuellement, mettre à jour $Out[n]$ suivant l'équation $Out[n] = (In[n] \setminus Kill[n]) \cup Gen[n]$.
3. Si l'un des ensembles In ou Out a été modifié, reprendre au point 2. Sinon, le point fixe est atteint.

EXEMPLE

Optimisations du processus de résolution Au point 2 de l'algorithme précédent, la mise à jour de l'ensemble $In[n]$ peut prendre en compte les mises à jours des prédécesseurs de n , si celles-ci ont déjà été faites. On accélère donc la découverte du point fixe en s'assurant que l'ordre de traitement des nœuds respecte au maximum l'orientation du graphe de flot de contrôle.

On peut aussi éviter de parcourir l'ensemble des nœuds du graphe à chaque changement, en travaillant avec un ensemble de nœuds à traiter de la manière suivante :

1. Initialiser l'ensemble des nœuds à traiter de sorte qu'il contienne tous les nœuds du graphe.
2. Tant que l'ensemble n'est pas vide, y prendre un nœud n :
 - (a) Mettre à jour les ensembles $In[n]$ et $Out[n]$.
 - (b) Si $Out[n]$ a été modifié, ajouter tous les successeurs de n qui n'y appartiennent pas déjà à l'ensemble des nœuds à traiter.
3. Une fois l'ensemble des nœuds à traiter vide, le point fixe est atteint.

Variantes Pour le calcul des variables vivantes, la méthode de résolution est la même. Attention cependant : comme les équations sont orientées différemment, l'ordre le plus judicieux pour la prise en compte des nœuds est différent.

Optimisations fonctionnelles

Voici les critères permettant d'appliquer les transformations illustrées au début du cours :

- La simplification des expressions constantes et l'utilisation des identités algébriques s'applique partout, à condition de ne pas supprimer d'effet de bord.
- La propagation des constantes s'applique en un nœud n où est lue une variable x dont on est certain qu'elle a été initialisée et pour laquelle exactement une définition est visible en entrée de n .
- L'élimination du code inutile s'applique à un nœud n contenant une affectation $x := e$ lorsque x n'est pas vivante en sortie de n (et que e ne produit pas d'effets de bord).
- L'élimination des sous-expressions communes s'applique à un nœud n calculant une expression disponible en entrée de n . Il faut alors inclure la sauvegarde de cette expression dans une variable temporaire au niveau de tout calcul de cette expression *accessible* en entrée de n .

Allocation de registres

Une des optimisations essentielles effectuées par les compilateurs consiste à utiliser au mieux les registres offerts par l'architecture cible (il s'agit donc d'une optimisation dépendante de la machine cible). Cette optimisation est particulièrement intéressante car les accès à la mémoire sont coûteux, alors qu'un calcul utilisant uniquement des registres pour stocker ses valeurs intermédiaires peut être extrêmement rapide.

De combien de registres a-t-on besoin ? Voici une expression qui nécessite deux registres sur l'architecture MIPS :

```
1*1 + 2*2
```

En effet, une fois le résultat de $1 + 1$ calculé, celui-ci doit être stocké dans un registre pendant le calcul du résultat de $2 + 2$.

De même, l'expression

```
(1*1 + 2*2) * (3*3 + 4*4)
```

nécessite trois registres : celui stockant le résultat de $1*1 + 2*2$ plus les deux nécessaires au calcul de $3*3 + 4*4$.

En suivant cette idée il est possible de créer des expressions nécessitant quatre, cinq, ou un nombre arbitraire de registres : il n'y a pas de limite au nombre de registres dont l'on peut avoir besoin, même pour une simple expression arithmétique.

Stratégie pour déterminer le nombre de registres nécessaires

- Pour commencer, on suppose avoir accès à un nombre arbitraire de *registres virtuels*, qu'on peut représenter comme des variables temporaires.
- On décompose les expressions pour expliciter chaque résultat intermédiaire, que l'on stocke dans un registre virtuel.
- On calcule les plages de vivacité de chaque variable représentant un registre virtuel.
- On en déduit un graphe d'interférence, dans lequel les nœuds sont les registres virtuels, et une arête est présente entre deux nœuds si et seulement si les deux variables correspondantes sont vivantes en un même point du programme.
- On cherche à colorier le graphe avec k couleurs, où k est le nombre de registres physiques disponibles.

- Si la coloration est possible : c'est gagné, l'ensemble des calculs peut être fait avec uniquement les registres.
- Sinon : il faut choisir des nœuds du graphe à « sacrifier », pour lesquels la valeur correspondante sera stockée sur la pile.