

TME 6 – Analyse et optimisation de programmes

Antoine Miné

1 Présentation

La performance d'exécution des programmes (en vitesse, en mémoire, etc.) est un facteur important. L'optimisation consiste à transformer un programme afin de le rendre plus efficace. Il est possible d'effectuer des optimisations à la main, sur le code source, mais cela a beaucoup d'inconvénients : cela prend du temps, rend le code moins lisible et risque d'ajouter des erreurs. C'est pourquoi les compilateurs modernes effectuent automatiquement des optimisations sophistiquées. Ces optimisations se basent sur des analyses préalables du code (dites analyses statiques, car ayant lieu avant l'exécution) et s'effectuent sur l'AST avant la génération de code.

Objectif. L'objectif du TME est d'ajouter une optimisation simple à ILP2 : l'intégration des fonctions. Considérons l'exemple suivant, contenant une fonction globale `f` d'argument `x`, et un corps avec une variable locale `y` et un appel à `f` :

```
1  function f(x) ( 2 * x + 1 );  
2  let y = 99 in  
3      z * f(y)
```

L'intégration des fonctions consiste à éviter un appel de fonction, coûteux, en le remplaçant par le corps de la fonction, ce qui donne le programme optimisé suivant :

```
1  function f(x) ( 2 * x + 1 );  
2  let y = 99 in  
3      z * (2 * y + 1)
```

Généralement, l'intégration des fonctions n'est qu'une étape d'optimisation parmi d'autres. Sur notre exemple, après l'intégration des fonctions, il devient possible de propager la valeur constante `99` de la variable `y` dans l'expression `z*(2*y+1)`, ce qui donne d'abord `z*(2*99+1)`, qui peut être ensuite simplifié en `z*199` par évaluation des parties constantes de l'expression. Après ces transformations, il devient même possible de supprimer la variable `y` et la fonction `f`, devenues inutiles, simplifiant le code généré. Dans ce TME, nous considérons uniquement la première étape, l'intégration des fonctions, et pas les optimisations suivantes (propagation des variables constantes, évaluation des sous-expressions constantes et élimination des variables et fonctions inutiles).

Buts :

- revoir les classes de l'AST en ILP2 ;
- travailler avec les visiteurs et les fabriques ;
- comprendre les notions d'analyse et de transformation statique de programmes ;
- effectuer des tests de régression.

2 Difficultés

Capture de variables. Considérons le programme :

```
1  function f(x) ( x + z );
2  z = 2;
3  let z = 12 in
4      f(z)
```

une intégration naïve de $f(z)$ donnerait :

```
1  function f(x) ( x + z );
2  z = 2;
3  let z = 12 in
4      z + z
```

qui donne pour résultat 24, alors que le programme original donnait 14. Le problème est que, au niveau de l'appel, la portée de z est celle de la déclaration locale `let z = 12` tandis que, au niveau de la définition de f , c'est celle de la variable globale $z = 2$. Notre solution à ce problème consistera à renommer toutes les variables locales pour éviter les conflits de nom. Ainsi `let z = 12 in f(z)` deviendra `let z0 = 12 in f(z0)` avant l'intégration de f . Après l'intégration, $f(z0)$ deviendra $z0 + z$, ce qui est correct.

Cycles d'appels. Un deuxième point délicat est le cas des fonctions récursives, comme par exemple :

```
1  function pow(x,n) ( if n < 0 then 1 else x * pow(x, n-1) );
2  pow(10,10)
```

Dans ce cas, une transformation qui remplacerait récursivement tout appel à `pow` par le corps de la fonction bouclerait, alors même que l'exécution du programme original termine. Notre solution à ce problème consistera à effectuer, avant l'intégration des fonctions, une analyse statique pour déterminer quelles fonctions interviennent dans des appels récursifs, et à interdire l'intégration de ces fonctions.

Tests de régression. Un test de régression consiste à déterminer si l'ajout d'une fonctionnalité n'a pas également provoqué l'ajout d'erreurs. Dans notre cas, les optimisations doivent être transparentes : le programme avant et le programme après optimisation doivent donner les mêmes résultats. Nous pouvons donc utiliser la batterie de tests existante pour valider chaque transformation.

3 Travail à réaliser

Les optimisations sont vues comme des transformations d'un AST en un autre AST. Ces transformations s'insèrent de manière transparente entre l'analyse syntaxique (texte vers AST) et la compilation (AST vers C) ou l'interprétation. Les analyses statiques, quant à elles, recueillent des informations par parcours de l'AST. Les transformations comme les analyses pourront donc être implantées à l'aide du motif visiteur. L'interface `IASTvisitor` sera en particulier utilisée. Afin de se simplifier la tâche, nous utiliserons le langage ILP2.

Note. Pour la compilation, nous aurions pu choisir également une transformation d'ASTC en ASTC, avec un `IASTCvisitor`. Nous avons choisi l'AST car il est plus simple que l'ASTC (moins de types de nœuds à gérer) et permet d'utiliser la transformation non seulement dans le compilateur mais également dans l'interprète.

3.1 Copie à l'identique

Notre première transformation se contente de reconstruire une copie de l'AST à l'identique. Il s'agit d'une copie profonde : chaque nœud visité commence par appeler récursivement le visiteur sur ses arguments, puis reconstruit un nouveau nœud avec ces copies en paramètre.

Cette transformation est bien sûr peu utile en elle-même. Elle servira par contre de patron de base à nos futures transformations. Il suffira alors d'hériter de cette classe en redéfinissant uniquement les visiteurs des nœuds à traiter spécialement.

Travail à réaliser. Dans un nouveau *package* `com.paracampus.ilp2.ilp2tme6`, vous programmerez une classe `CopyTransform<Data>` implantant l'interface `IASTvisitor<IAST,Data,CompilationException>` et effectuant une copie profonde de l'AST. La création des nœuds se fera par une fabrique `IASTFactory`, passée en argument au constructeur de `CopyTransform`. En plus des visiteurs de `IASTvisitor`, qui permettent la copie des nœuds expressions, il sera nécessaire de fournir un point d'entrée prenant en argument un programme complet et retournant la copie du programme (après avoir copié chaque déclaration de fonction du programme, puis le corps du programme). Cette fonction pourra avoir la signature :

```
public IASTprogram visit(IASTprogram iast, Data data);
```

Pour l'instant, l'argument `data` du visiteur n'est pas utilisé, mais il pourra l'être dans les sous-classes de `CopyTransform`. Nous avons donc laissé son type comme paramètre du générique, `Data`.

Vous ajouterez une classe `ILPMLOptimizingParser` (héritant de `ILPMLParser`) qui applique cette transformation dans la méthode `getProgram`, puis retourne le programme transformé.

Vous effectuerez enfin des tests de régression pour vérifier que l'interprétation et la compilation de la copie de l'AST donnent les mêmes résultats que celles de l'AST original. N'oubliez pas d'ajouter ces tests à l'intégration continue puis de vérifier, après un *push*, qu'ils fonctionnent correctement sur le serveur GitLab du cours.

3.2 Renommage des variables (alpha-conversion)

Afin d'éviter le phénomène de capture de variable décrit en section 2, nous commençons par nous assurer que toutes les variables locales et globales ont des noms différents. Pour cela, nous allons effectuer une transformation, appelée *alpha-conversion*, consistant à donner un nouveau nom unique à chaque variable liée, c'est à dire à chaque variable introduite par un nœud `IASTbloc` ou un argument formel de fonction d'un nœud `IASTfunctionDefinition`, sans toucher aux variables globales ni au nom des fonctions globales.

Par exemple :

```
1 function f(x) ( let x = x + 2 in 2 * x );
2 x = 12;
3 let x = 3 * x in
4     f(3 * x)
```

deviendra :

```
1 function f(x_1) ( let x_2 = x_1 + 2 in 2 * x_2 );
2 x = 12;
3 let x_3 = 3 * x in
4     f(3 * x_3)
```

où `x_1`, `x_2` et `x_3` sont les nouveaux noms uniques des différentes variables locales appelées `x`, tandis que la variable globale `x` garde son nom original.

Travail à réaliser. Vous implanterez une classe `RenameTransform` effectuant cette transformation. Cette classe héritera de `CopyTransform` et redéfinira le nombre minimal de méthodes possibles (3 devraient suffire). Afin de donner un nom unique à chaque variable, vous pourrez utiliser un compteur global, incrémenté à chaque nouvelle déclaration de variable. Enfin, la classe `INormalizationEnvironment` sera utile pour représenter un environnement de renommage, qui sera passé dans l'argument `data` du visiteur (cf. la normalisation de l'AST vers l'ASTC).

Vous modifierez l'interprète et le compilateur pour utiliser cette nouvelle transformation.

Vous effectuerez des tests de régression, en local et en intégration continue sur le serveur GitLab du cours : l'exécution du programme transformé doit donner le même résultat que celle du programme original. Vous vérifierez également visuellement que le code C généré par le compilateur utilise bien les nouveaux noms de variable.

3.3 Détection des fonctions récursives

Nous déterminons maintenant les fonctions récursives. Une première étape consiste à calculer le graphe d'appel. Il s'agit d'une table `Map<String,Set<String>>` qui associe, à chaque nom de fonction, l'ensemble des noms des fonctions qu'elle peut appeler. Pour chaque déclaration de fonction, vous exécuterez un visiteur sur son corps afin de récupérer l'ensemble des noms des fonctions appelées et mettrez à jour la table. Un tel appel de fonction est matérialisé par la présence d'un nœud `IASTinvocation`.

La deuxième étape consiste à déterminer les *cycles* de ce graphe. Toute fonction apparaissant dans un cycle du graphe sera considérée comme récursive et ne sera pas intégrée. Vous pourrez utiliser un algorithme classique d'exploration de graphe en profondeur. Ce mécanisme permet donc de détecter non seulement les appels récursifs directs (`f` appelle `f`), mais aussi les fonctions mutuellement récursives (`f` appelle `g` et `g` appelle `f`).

Travail à réaliser. Vous implanterez une classe `CallAnalysis` qui calcule le graphe d'appel. Le calcul du graphe d'appel et la détermination des cycles peut, par exemple, se faire dans le constructeur de la classe. La classe proposera également une méthode publique `boolean isRecursive(IASTvariable f)` qui permettra à un client d'accéder au résultat de l'analyse. La méthode retournera `true` si la fonction donnée en argument apparaît dans un cycle d'appels récursifs.

Vous développerez des tests unitaires pour vérifier que, pour quelques programmes bien choisis, `isRecursive` donne le résultat attendu. Faites un *push* et vérifiez que vos tests fonctionnent sur le serveur GitLab du cours.

3.4 Intégration des fonctions (*inlining*)

Étant donnée une définition de fonction :

```
function f(x,y) expr
```

l'intégration de `f` transformera un appel `f(expr1,expr2)` en un bloc :

```
let x, y = expr1, expr2 in expr
```

Ce bloc commence par lier les arguments formels `x` et `y` à leur valeur réelle `expr1` et `expr2`, avant d'exécuter le corps de la fonction.

Travail à réaliser. Vous implanterez une classe `InlineTransform` qui effectue cette transformation. Cette classe fera naturellement appel à `CallAnalysis` et à `RenameTransform` pour éviter les écueils liés aux fonctions récursives et à la capture de variables. Vous ajouterez cette transformation au compilateur et à l'interprète. Enfin, vous testerez cette transformation. En particulier, vous ajouterez des tests mettant en valeur l'absence de problèmes liés à la récursivité ou à la capture de variables. Après avoir fait fonctionner vos tests en local, ajoutez-les à l'intégration continue, faites un *push* et vérifiez qu'ils fonctionnent également sur le serveur GitLab du cours.

4 Pour aller plus loin

Si vous avez le temps, vous pourrez ajouter l'optimisation suivante : la suppression des variables et fonctions inutiles. Il s'agit de supprimer du programme les déclarations de variables locales, de variables globales et de fonctions globales qui ne sont pas utilisées dans le programme. Attention, si une variable globale n'est utilisée que dans une fonction globale jamais appelée, cette variable globale ne doit pas être considérée comme utilisée. Comme pour l'intégration des fonctions, cette optimisation nécessite deux étapes : d'abord une analyse statique, déterminant les variables et fonctions inutiles, suivie d'une étape de reconstruction de l'AST. Cette optimisation sera particulièrement efficace si elle est effectuée après l'intégration des fonctions, puisque celle-ci supprime de nombreux appels de fonctions.

5 Rendu

Comme pour les TME précédents, vous effectuerez un rendu en vous assurant que tout le code développé a été envoyé sur le serveur GitLab (*push*) dans votre *fork* d'ILP2. Vous vous assurerez que les tests d'intégration continue développés lors de ce TME ont été configurés (fichier `.gitlab-ci.yml`) et fonctionnent sur le serveur. Vous ajouterez un tag « rendu-initial-tme6 » en fin de séance, puis un tag « rendu-final-tme6 » quand le TME est finalisé.

Dans la section *Release note* associée à votre tag final, vous indiquerez quelles optimisations vous avez implantées, en particulier si vous avez réalisé des extensions proposées à la question précédente.