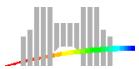
Master Informatique Fondamentale - M1 Compilation

Représentations intermédiaires

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

11 février 2007



Deux rôles :

- Base de connaissances sur le programme source, plus facile à interroger que le texte du programme.
- Représentation compacte de l'avancement de la compilation (l'état actuel du programme compilé).
- Au fur et à mesure de la compilation, la RI s'enrichit (analyse) et se modifie (transformation et optimisation).
- La RI est en général une structure binaire, mais il faut être capable de l'afficher (mise au point du compilateur) et de la sérialiser (communication entre phases).
- Dans un compilateur bien fait, on dote la RI d'un API permettant de la construire, de la modifier et de l'exploiter, et qui encapsule les détails techniques.

Combien de représentations intermédiaires ?

- Il est naturel de changer de représentation au fur et à mesure que la compilation avance : les questions posées ne sont plus les mêmes.
- Inconvénients :
  - Maintien de la cohérence
  - Traductions
- On peut préférer une unique RI.
- Il est difficile de concevoir une RI universelle.

Arbre de syntaxe abstraite (AST)

- Version optimisée de l'arbre d'analyse syntaxique.
- Avantages :
  - Pas de perte d'information.
  - Il est très facile d'ajouter des informations au fur et à mesure des analyses.
- Inconvénients :
  - Certains renseignements sont difficiles à obtenir : par exemple, le successeur d'une instruction.
  - Les informations qui concernent plus d'une instruction (les dépendances) sont difficiles à représenter.
  - Il est délicat de modifier la structure d'arbre (par exemple, découper une instruction en deux) surtout si le langage est fonctionnel.

Arbres Lisp

Au lieu d'utiliser des structures de données complexes, on peut imiter les arbres de Lisp :

```
type lisp = Nil
          | Atom of string
          | Cons of lisp * lisp
;;
```

- Tout arbre peut être représenté en Lisp
- Simplification des algorithmes (substitution, unification)
- Avantage / inconvénient : pas de contrôle de type.
- En fait, les atomes jouent en partie le rôle de types manifestes

Graphe de contrôle

Permet de représenter l'organigramme du programme.

- Sommets : les instructions du programme.
- Un arc de *a* vers *b* s'il est possible d'exécuter *b* juste après avoir exécuté *a*.
- Un seul arc sortant sauf pour les tests.
- Le graphe de contrôle doit avoir un point d'entrée et un point de sortie uniques.
- Peut être indépendant de l'AST.
- Peut être également "transfilé" dans l'AST. Il suffit de prévoir des pointeurs supplémentaires dans les nœuds de l'arbre.
- On peut également convenir que les instruction sont sur les arcs et que les sommets sont les états stables entre instructions.

Construction par grammaire attribuée

```
program : statement;

statement : STAT
          | BEGIN statements END;

statements : statement | statements statement ;
```

Le non terminal `program` (axiome de la grammaire) sert uniquement à marquer le début et la fin du graphe.
Le terminal `STAT` représente les instructions non analysées.

Construction du graphe de contrôle, II

On commence par synthétiser deux attributs `first` et `last` pour chaque objet.

```
program : statement
        {$1.first = $1.last = NULL}

statement : STAT
          {$0.first = $0.last = $1}
          | BEGIN statements END
          {$0.first = $1.first; $0.last = $1.last}

statements : statement
           {$0.first = $1.first; $0.last = $1.last}
           | statements statement
           {$0.first = $1.first; $0.last = $2.last;
            $1.last.succ = $2.first; $2.first.pred = $1.last}
```

On construit ensuite le graphe (attributs `pred`, `succ` par une analyse descendante.

Exercice : compléter la grammaire (boucles, conditionnelles,

Bloc de base

- Bloc de base : suite d'instruction sans point d'entrée intermédiaire ni sortie prématurée.
- La grammaire ci-dessus n'engendre qu'un seul bloc de base.
- Les choses sont plus complexes quand le langage a des conditionnelles, des boucles, et surtout des *sauts* : `break`, `continue`, `goto`.
- Les blocs de base sont les unités de travail des optimiseurs.
- En général, on compacte le graphe de contrôle : un sommet = un bloc de base.

Code trois adresses

Une représentation linéaire est très proche d'un programme en langage machine. En général, on limite la représentation linéaire aux bloc de base (mais on peut également mettre tout le programme en linéaire).

- Chaque élément de la représentation est une opération qui peut porter sur une, deux ou trois variables (le plus fréquent).
- Comme la complexité des instruction originales peut être aussi grande q'on le veut, pour passer en trois adresses il faut introduire des temporaires ou *registres virtuels*.
- Le passage en code trois adresses peut se faire au moyen d'une grammaire attribuée, voir plus haut.
- On peut également envisager du code à une adresse (machine à pile) et du code à deux adresses :

```
x := x + y;
```

Analyse du graphe de contrôle

En se limitant au graphe de contrôle ou à une représentation linéaire, on perd une information importante : les boucles.
Exemple : le bytecode de Java.
Reconstituer les boucles à partir du graphe de contrôle.

- Un sommet *a domine* un sommet *b* si tout chemin allant du point d'entrée jusqu'à *b* passe par *a*.
- Toute boucle a un point d'entrée qui domine toutes les instructions de la boucle.
- L'arc de retour a la propriété que sa fin domine son début.
- Calculer la relation de domination
- Identifier les arcs de retour et les points d'entrée.
- Le corps de boucle est l'ensemble des instructions qui sont à la fois prédécesseurs et successeurs du point d'entrée.

Calcul des dominateurs

Principe

- Tout sommet a se domine lui-même.
- d domine a si d domine tous les prédécesseurs de a.
- On initialise $D(a)$ à l'ensemble des sommets.
- On itére $D(a) := \{a\} \cup \bigcup_{p \in Pred(a)} D(p)$.
- On montre que les ensembles $D(a)$ se réduisent à chaque itération ce qui implique la convergence.

Static Single Assignment Form

Un programme est en forme SSA si chaque variable *scalaire* n'est moidfiée qu'une seule fois dans le *texte* du programme.

- Certaines optimisations sont plus faciles sur la forme SSA (au moins dans un bloc de base) parce qu'il y a bijection entre les noms et les valeurs. Exemple : la détection des calculs redondants.
- Pour passer en forme SSA :
  - Renommer (numéroter) systèmatiquement les variables au fur et à mesure de leur écriture (à gauche d'une affectation).
  - Remplacer une variable lue par sa plus proche écriture antérieure ; s'il y a ambiguïté, introduire une ϕ-fonction (un multiplexeur).

Forme SSA, Exemple

$x := \ldots;$
$y := \ldots;$
while $x < 100$ **do**
    $x := x + 1;$
    $y := y + x;$
end

$x_0 := \ldots;$
$y_0 := \ldots;$
while $(x_1 := \phi(x_0, x_2)) < 100$ **do**
    $x_2 := x_1 + 1;$
    $y_1 := \phi(y_0, y_2);$
    $y_2 := y_1 + x_2;$
end
$x_3 := \phi(x_0, x_2);$
$y_3 := \phi(y_0, y_3);$

Représentations actives, exemples

- Une base de donnée est active si elle est capable de raisonner logiquement à partir de faits connus.
- Exemple des relations de parenté, (DATALOG).
- De même, une RI est active si elle élabore des informations à partir des faits connus.
- Scenario ;
  - $x \leftarrow u + v$
  - $u \leftarrow 3$
  - $v \leftarrow 1$
  - x ? réponse 4.
- La méthode peut être utilisée pour manipuler les expressions arithmétiques et booléennes.

Formes normales

Outil de base : un algorithme de normalisation.

- Etant donné un système de règles de calcul, une forme normale est une expression qui ne peut plus être réécrite.
- Le système est confluent si la forme normale d'une expression est unique.
- L'existence d'un algorithme de normalisation implique la décidabilité de l'égalité. Il n'y a donc pas de forme normale pour les programmes.
- Mais il existe des formes normales pour les polynomes et les expressions booléennes (CNF, DNF, OBDD).

Bilan

- La conception de la RI a une grande influence sur la qualité et la facilité d'écriture du compilateur.
- En général, on a besoin de plusieurs RI, et chaque phase du compilateur peut être vue comme un transformateur de RI.
- Les phases initiales travaillent sur l'AST.
- Les phases finales utilisent une représentation linéaire, très proche du langage machine.
- Mais il est tentant d'utiliser une représentation hybride :
  - Arbre de syntaxe abstraite
  - Graphe de contrôle reliant entre eux les nœuds de l'AST
  - Code trois adresse comme expansion des nœuds de l'AST.
  - Formes normales pour les expressions et les calculs sur scalaire dans les blocs de base.