CSC4251_4252 : Représentation Intermédiaire

Pascal Hennequin, J. Paul Gibson, Denis Conan

Novembre 2024

Sommaire

1 Objectif

- **Objectif génie logiciel**
  - Modularité du compilateur $\Longrightarrow$ Interface entre :
    - Partie avant dépendante du langage source
    - Partie arrière dépendante de la machine cible (assembleurs ou autres)
- **Objectif programmation**
  - Étape intermédiaire pour la phase « Génération »
  - Génération = traduction de l'AST vers code machine
    - Linéarisation du code
    - Génération de l'assembleur
    - Allocation mémoire
    - Optimisations : allocations des registres, évaluation des expressions, ...

2 Principe I

Génération de la représentation intermédiaire

- Convertir les nœuds de l'AST en utilisant un nombre réduit d'« instructions »
- Utiliser des instructions « intermédiaires » entre le langage source et le langage cible
- Linéarisation (canonisation) :
  - Traduire la sémantique d'exécution sous forme de séquence d'instructions
  - Traduire en particulier les structures algorithmiques et les appels de fonction sous forme de code séquentiel avec des sauts
  - Les expressions peuvent être complètement linéarisées (« classique ») ou conserver la structure d'arbre (« moderne »)
  - N.B. : On fait implicitement de la « couture d'AST » et donc possibilités de faire de l'analyse sémantique dynamique après la forme intermédiaire

2 Principe II

Représentation Intermédiaire

- Instructions
  - Affectations ou copie
  - Labels pour sauts et appels de fonction
  - Sauts inconditionnels et conditionnels
  - Expressions (arbres ou linéarisées)
  - Gestion des Appels : ...
- Variables
  - Héritées de l'AST
  - Ajoutées pour la forme intermédiaire : évaluation des expressions, gestion d'appels, ...
- Informations administratives
  - Définition des symboles IR : variables, labels, constantes/immediate
  - ...

3 Code à trois adresses

- **Programme**
  - Séquence d'instructions
  - Linéarisation complète des expressions
- **Instruction**
  - Une opération et zéro à trois « adresses » [OP, X, Y, Z]
  - « adresses » ou variables intermédiaires $Z = X\ OP\ Y$
    - Variables du programme source (ou nom de fonction, ou ...)
    - Variables temporaires pour évaluation des expressions
    - Labels pour des sauts
    - Valeurs constantes
- **Exemple :**

$A = 2 \times x + 1$

$[\times, 2, x, t_1]$
$[+, t_1, 1, t_2]$
$[=, t_2, , A]$

$t_1 = 2 \times x$
$t_2 = t_1 + 1$
$A = t_2$

4 Représentation pour MiniJAVA I

- **Représentation Intermédiaire : classe `intermediate.IR`**
  - Programme = séquence de `IRquadruple`
    - Construction itérative par ajout de `IRquadruple`
  - Création de variables intermédiaires :
    - `IRvariable newTemp(String method)`, `IRvariable newConst(int value)`
    - `IRvariable newLabel()`, `IRvariable newLabel(String label)`
- **Instructions : paquetage `intermediate.ir`**
  - `abstract IRquadruple = [op,arg1,arg2,result]`
  - `main.EnumOper op` : opérateur, uniquement nœuds `*Assign*`
  - `IRvariable result, arg1, arg2` : variables intermédiaires
  - Les variables de l'AST (`semantic.symtab.infoVar`) implémentent `IRvariable`

4 Représentation pour MiniJAVA II

- **Instructions (suite)**
  - Affectation : `QCopy`
  - Labels : `QLabel` et `QLabelMeth`
  - Sauts : inconditionnel avec `QJump` et conditionnel avec `QJumpCond`
  - Gestion d'appel : `QCall`, `QCallStatic`, `QParam`, `QReturn` (et `QlabelMeth`)
  - Expressions : `QAssign`, `QAssignUnary`, `QNew`
  - Expressions avec Tableau : `QAssignArrayFrom`, `QAssignArrayTo`, `QNewArray`, `QLength`

5 Traduction AST vers IR I

- `Génération de la représentation intermédiaire`
  - Classe `intermediate.Intermediate`
  - Visite récursive en profondeur de l'AST
  - Utilisation d'un attribut synthétisé de type `IRvariable`
    - Requis pour les nœuds expressions de l'AST (`Expr*`)
    - Contient la variable (temporaire ou constante ou AST) utilisée dans la représentation intermédiaire pour stocker le résultat de l'expression
    - Les méthodes `setVar()` et `getVar()` affecte et retourne l'attribut `IRvariable` associé à chaque nœud de l'AST
  - Traduction et linéarisation
    - Concerne l'ensemble des nœuds de l'AST, à l'exception de quelques nœuds uniquement déclaratifs (`Klass`, `Formal`, etc)

5 Traduction AST vers IR II

Schémas de traduction

- Expressions
  - N.B. : toutes les « traduction(exp) » exécute un `setVar(exp,...)`

```
/* ExprLiteralInt */
42
```

```
setVar node, newConst(42))
```

```
/* ExprLiteralBool */
false
```

```
setVar node, newConst(0)
/* ou 1 pour true */
```

```
/* ExprOpBin */
exp1 + exp2
```

```
traduction(exp1)
traduction(exp2)
setVar node, newTemp()
QAssign +, getVar(exp1), getVar(exp2), getVar(node)
```

- ...

5 Traduction AST vers IR III

Schémas de traduction(suite)

- Déclaration et appel de méthode

```
/* Method */
xxx(...) {
  contenu
  Return exp }
```

```
QLabelMeth newLabel(‘‘xxx’’)
traduction(contenu)
traduction(exp)
QReturn getVar(exp)
```

```
/* ExprCall */
exp.xxx(arg1,arg2,..)
```

```
traduction(exp)
traduction(arg1)
/* ... args suivants */
QParam getVar(exp) // this
QParam getVar(arg1)
/* ... args suivants */
QCall newLabel(‘‘xxx’’), newConst(nb_args)
```

5 Traduction AST vers IR IV

Schémas de traduction (fin)

Instructions

```
/* StmtPrint */
System.out.println(exp)
```

```
traduction(exp)
QParam getVar(exp)
QCallStatic newLabel(``_system_out_println''), ``1''
```

```
/* StmtAssign */
X = exp
```

```
traduction(exp)
setVar(node, lookup(``X''))
QCopy getvar(exp), getvar(node)
```

```
/* StmtWhile */
while (exp) inst
```

```
QLabel L1
traduction(exp)
QJumpCond L2, getVar(exp)
traduction(inst)
QJump L1
QLabel L2
```

```
L1 = newLabel()
L2 = newLabel()
```