CSC4251_4252 : Représentation Intermédiaire

Pascal Hennequin, Denis Conan, J. Paul Gibson

Novembre 2024

Sommaire

1 Exemple illustratif I

- Exemple extrait de MiniJAVA : `Test102.txt`

```
1 class Test102 {
2   public static void main(String[] args) {
3     System.out.println(2 + 6 * 7 - (5 - 3));
4   }
5 }
```

- Représentation intermédiaire linéarisée « Code à 3 adresses »

```
1 "main"                | QLabel[null, main, null, null]
2 t_0 = 6 * 7           | QAssign[*, c_6, c_7, t_0]
3 t_1 = 2 + t_0         | QAssign[+, c_2, t_0, t_1]
4 t_2 = 5 - 3           | QAssign[-, c_5, c_3, t_2]
5 t_3 = t_1 - t_2       | QAssign[-, t_1, t_2, t_3]
6 param t_3             | QParam[null, t_3, null, null]
7 call _println<1>      | QCallStatic[null, _println, c_1, null]
```

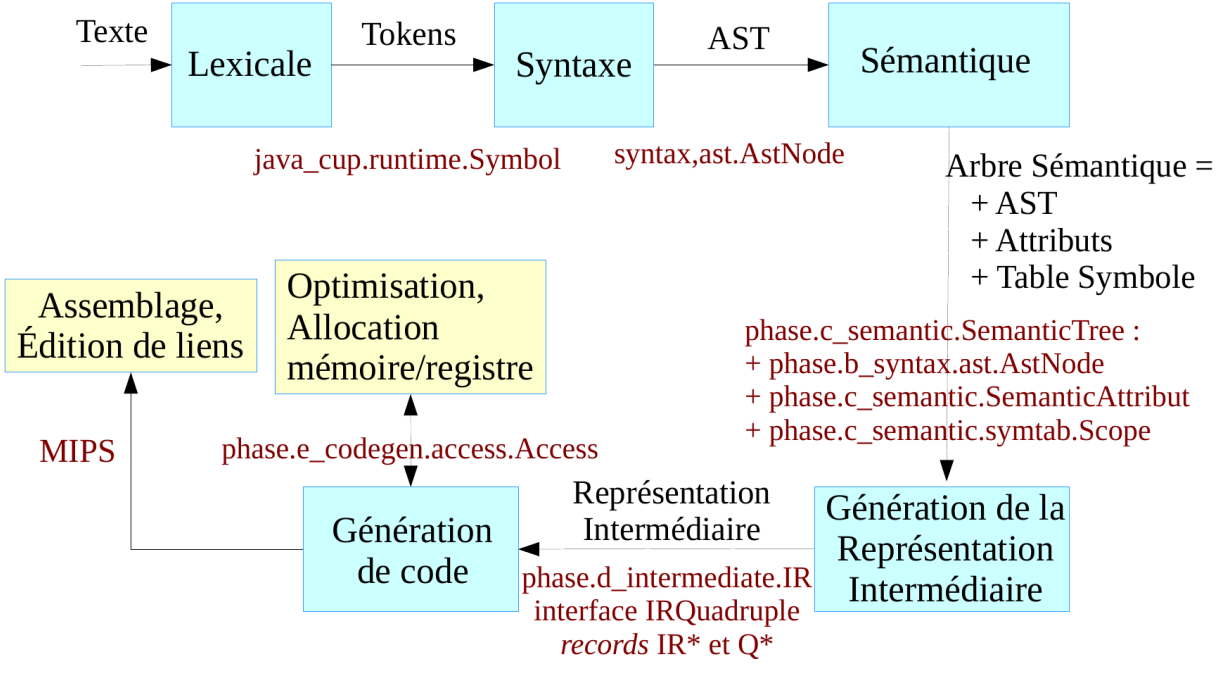
1 Exemple illustratif II

- Représentation intermédiaire *Byte Code* JAVA

```
1  class Test102 {
2   Test102();
3    Code:
4     0: aload_0
5     1: invokespecial #1  // Method java/lang/Object."<init>":()V
6     4: return
7
8   public static void main(java.lang.String[]);
9    Code:
10    0: getstatic     #7  // Field java/lang/System.out:Ljava/io/PrintStream;
11    3: bipush        42
12    5: invokevirtual #13 // Method java/io/PrintStream.println:(I)V
13    8: return
14 }
```

2 Objectifs I

- **Objectif génie logiciel : Modularité (frontale/finale)**
  - Fin de la partie frontale = production d'une représentation intermédiaire
    - Dans les cas simples, détails relatifs au langage source cantonnés à la partie frontale
  - Partie dite « finale » = à venir, production de code assembleur



2 Objectifs II

Objectif optimisation

- Selon la forme de la représentation intermédiaire, d'autres optimisations
  - Par exemple, générer une représentation intérmédiaire de l'expression booléenne $expr_1 \wedge expr_2$ telle que : pas d'évaluation de $expr_2$ si $expr_1 = \text{faux}$
- Certaines optimisations sont plus faciles à faire dans la représentation intermédiaire
  - Par exemple, dans une représentation intermédiaire linéarisé, détecter le code « mort » qui se trouve entre la dernière instruction `return` et le retour d'appel

3 Principe

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

Représentation Intermédiaire

- Instructions : affectations ou copie, sauts et appels de méthodes/fonctions, etc.
- Variables
  - Héritées de l'AST
  - Ajoutées pour la forme intermédiaire : évaluation des expressions, gestion d'appel, etc.
- Informations administratives
  - Définition des symboles : variables, labels, constantes/immediate
  - Etc.

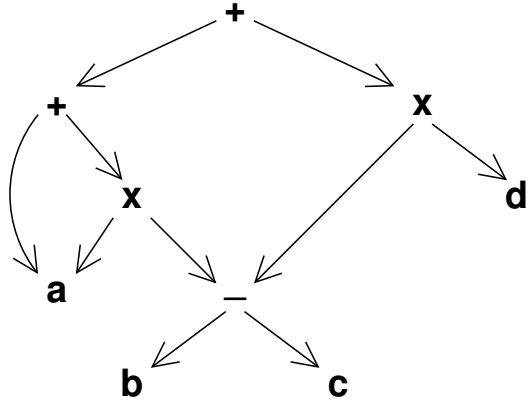
4 Formes de la représentation intermédiaire

- École classique
  - « La plus simple » avec par exemple le code à trois adresses
    - Linéaire avec des sauts
- École moderne
  - « Plus évoluée » avec un graphe orienté acyclique (*Direct Acyclic Graph*, DAG)
- Exemple : « $a + a \times (b - c) + (b - c) \times d$ »

École classique

$t_1 = b - c$	$[-,\ b,\ c,\ t_1]$
$t_2 = a \times t_1$	$[\times,\ a,\ t_1,\ t_2]$
$t_3 = a + t_2$	$[+,\ a,\ t_2,\ t_3]$
$t_4 = t_1 \times d$	$[\times,\ t_1,\ d,\ t_4]$
$t_5 = t_3 + t_4$	$[+,\ t_3,\ t_4,\ t_5]$

École moderne



5 Code à 3 adresses

C'est la solution que nous choisissons pour sa simplicité.

5.1 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
- Avec la forme linéarisée, une analyse sémantique comportementale peut permettre de faire des optimisations
  - Exemple d'optimisation difficile à faire à partir de l'AST décoré : entremêlage
    - Variable « a » dans un registre lors de l'appel de méthode « `m(a)` »
    - Incrémenter « a » tout de suite après le retour plutôt que d'avoir à recharger la variable plus tard dans un registre

```
int a = 0;
while (a < 10) {
  m(a);
  ... // pas de modification de a
  a++;
}
```

```
int a = 0;
while (a < 10) {
  m(a);
  a++;
  ... // pas de modification de a
}
```

5.2 Forme des Instructions

- **Une opération, et zéro à trois « adresses »** [OP, X, Y, Z]
- **Adresses**
  - Noms de variables
    - par commodité, noms extraits des programmes source
    - dans l'idéal, devraient être calculés à partir des identificateurs de la table des symboles pour être uniques
  - Noms d'étiquettes/labels pour les sauts produits par le compilateur
    - par commodité, noms extraits des programmes source pour les méthodes
    - génération par le compilateur d'identificateurs uniques pour les sauts introduits dans la traduction des structures de contrôle (*if*, *while*, etc.)
  - Constantes (*immediate*s de l'assembleur)
  - Variables temporaires produites par le compilateur
    - Typiquement lors de la linéarisation des expressions

5.3 Ensembles des Instructions sélectionnées

Instructions

- Affectations de la forme $Z = X\ OP\ Y$
- Affectations de la forme $X = OP\ Y$
- Instruction de copie de la forme $X = Y$
- Instruction de copie utilisant des variables indicées, de la forme $X = Y[i]$ / $X[i] = Y$
- Branchement inconditionnel (saut à une étiquette/label) aller à E
- Branchement conditionnel [1]
  si X aller à E
  sifaux X aller à E
- Appel (et retour de procédure) [2]
  arg x_1
  ...
  arg x_n
  appeler p, n
- ...

1. La forme « `si X aller à E` » n'existera pas dans MiniJAVA.
2. Cf. cours suivant pour la convention d'appel.

5.4 Forme à affectation unique

- **Affectation statique unique (*Static single assignment*, SSA)**
  - Pour facilité certaines optimisations
  - Toutes les affectations se font vers des variables distinctes

    $y = 1\,;$ $y = 2\,;$ $x = y\,;$ devient $y_1 = 1\,;$ $y_2 = 2\,;$ $x_1 = y_2$
  - Déclaration spéciale ϕ pour les jointures après des chemins différents

    if (c) $u = -1\,;$ else $u = 1\,;$ $v = u \times a$ devient if (c) $u_1 = -1\,;$ else $u_2 = 1\,;$ $u_3 = \phi(u_1, u_2)$ $v_1 = u_3 \times a$
  - Intérêt : analyse d'atteinte des définitions facilitée (p.ex. y_1 jamais utilisée)
- Dans MiniJAVA, on n'est peut-être pas très loin, mais on n'y est pas

6 Représentation Inter. pour MiniJAVA I

- ***Record*** `intermediate.IntemediateRepresentation`
  - Programme = séquence de `IRQuadruple` (op, arg1, arg2, res)
    - Construction itérative par ajout de `IRQuadruple`
- **Création de variables intermédiaires, y compris les étiquettes/labels**
  - Méthodes *helpers* dans le visiteur `Intermediate`
    - Variables temporaires : `IRVariable newTemp(String method)`
      - Variable locale à une méthode, c.-à-d. dans la portée de la méthode englobante
    - Constantes : `IRVariable newConst(int value)`
    - Étiquettes/labels :
      - Pour des sauts de contrôle : `IRVariable newLabel()`
      - Pour les méthodes : `IRVariable newLabel(String label)`

6 Représentation Inter. pour MiniJAVA II

- **Instructions : paquetage `intermediate.ir`**
  - Interface `IRQuadruple` = {`op()`, `arg1()`, `arg2()`, `result()`}
    - `op()` : un énumérateur de `compil.EnumOper`
    - `arg1()`, `arg2()`, `result()` : une variable intermédiaire `IRVariable`
      - Les variables de l'AST (`phase.c_semantic.symtab.infoVar`) implémentent `IRVariable`
  - *Records* implémentant l'interface *IRQuadruple*
    - Affectation : `QCopy(IRVariable,IRVariable)`
    - Étiquettes : `QLabel(IRLabel)` et `QLabelMeth(IRLabel,IRConst)`, avec `IRConst` qui est le nombre de paramètres, qui comprend `this`
    - Saut inconditionnel avec `QJump(IRLabel)`
      et saut conditionnel `siFaux` avec `QJumpCond(IRLabel,IRVariable)`

6 Représentation Inter. pour MiniJAVA III

- Gestion d'appel :
  - Appel de méthode d'instance avec `QCall(IRLabel,IRConst,IRVariable)`
  - Appel de méthode de classe avec `QCallStatic(IRLabel,IRConst)`
  - Passage de paramètre avec `QParam(IRVariable)`
  - Retour d'appel avec `QReturn(IRVariable)`
- Gestion des expressions :
  - Opérateur binaire avec `QAssign(EnumOper,IRVariable,IRVariable,IRVariable)`
  - Opérateur unaire avec `QAssignUnary(EnumOper,IRVariable,IRVariable)`
  - Instanciation d'objet avec `QNew(IRLabel,IRVariable)`
- Gestion des expressions avec Tableau : `QLength(...)` pour `res=length(op1)`
  `QAssignArrayFrom(...)` pour `res=op1[op2]`,
  `QAssignArrayTo(...)` pour `res[op2]=op1`,
  `QNewArray(...)` pour `res=new op1[op2]`

7 Traduction AST vers Représentation Inter. I

- `Intermediate` construit un objet `IntermediateRepresentation` =
  - + Programme en tant que séquence de `IRQuadruples`
  - + Racine de l'AST (alloc. mémoire, attr. classe avec attr. classes parentes)
  - + Liste des variables temporaires + Liste de constantes
  - + Liste d'étiquettes/labels
- Utilisation d'un attribut synthétisé de type `IRVariable`
  - Contient la variable (temporaire ou constante ou AST) utilisée dans la représentation intermédiaire pour stocker le résultat de l'expression
  - La méthode `setVar` (resp. `getVar`) affecte (resp. renvoie) l'attribut `IRVariable` associé à chaque nœud de l'AST
    - Si la variable recherchée n'est pas trouvée alors `getVar` renvoie une nouvelle « variable indéfinie »
- Utilisation d'un attribut hérité `currentMethod`
  - Une nouvelle variable temporaire est créée dans la portée d'une méthode

7 Traduction AST vers Représentation Inter. II

- **Traduction du nœud de type `KlassMain`** : donné ou voir la solution
- **Traduction des nœuds de type `Expr*`**
  - Approche choisie pour MiniJAVA : utilisation de variables temporaires
    - Donc, toutes les « `traduction(exp)` » exécute un « `setVar(exp,...)` »
  - Autre approche possible : utilisation d'une pile
    - C'est l'approche choisie pour le *byte code* JAVA

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

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

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

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

```
/* ExprIdent */
v
```

```
setVar(node, lookupVar dans la portée courante, c.-à-d. de node)
```

7 Traduction AST vers Représentation Inter. III

```
/* ExprOpUn */
−exp
```

```
traduction(exp)
setVar(node, newTemp()) /* nouvelle variable temporaire */
ajout QAssign avec op(), getVar(exp), getVar(node)
```

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

```
traduction(exp1)
traduction(exp2)
setVar(node, newTemp()) /* nouvelle variable temporaire */
ajout QAssign avec op(), getVar(exp1), getVar(exp2), getVar(node)
```

```
/* ExpNew */
new A()
```

```
setVar(node, newTemp()) /* nouvelle variable temporaire */
ajout QNew avec label = nom classe de l'objet, getVar(node)
```

7 Traduction AST vers Représentation Inter. IV

- **Déclaration et appel de méthode**

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

```
ajout QLabelMeth newLabel(‘‘xxx’’), newConst(nb args en comptant this)
save = currentMethod
currentMethod = ‘‘xxx’’
traduction(contenu)
traduction(exp)
currentMethod = saved
QReturn getVar(exp)
```

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

```
traduction(exp)
traduction(arg1)
/* ... args suivants */
ajout QParam getVar(exp) /* ne pas oublier this ici et dans nb_args */
ajout QParam getVar(arg1)
/* ... args suivants */
setVar(node, newTemp()) /* nouvelle variable temporaire */
ajout QCall newLabel(‘‘xxx’’), newConst(nb_args), getVar(node)
```

7 Traduction AST vers Représentation Inter. V

- **Instructions**

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

```
traduction(exp)
ajout QParam getVar(exp)
ajout QCallStatic newLabel(``_println''), newConst(1)
                  /* _println est dans le Runtime MiniJAVA */
```

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

```
traduction(exp)
info = lookupVar de ``X'' dans node
ajout QCopy info, getvar(node)
```

```
/* StmtBlock */
{ contenu }
```

```
defaultVisit de node
```

7 Traduction AST vers Représentation Inter. VI

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

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

```
/* StmtIf */
if (exp) inst1 else inst1
```

```
L1 = newLabel()
L2 = newLabel()
traduction(exp)
ajout QJumpCond L1, getVar(exp)
traduction inst1
ajout QJump L2
ajout QLabel L1
traduction inst2
ajout QLabel L2
```