Partie VIII : Production de code II

Dans cette partie, nous examinons la traduction des conditions et des énoncés de contrôle.

Comparaisons et sauts

- Soit l'énoncé

```
if (x == y) expression ...
```

- On le traduit de la manière suivante :

```
  CMP R1, x, y  // R1 := x -- y
  BNE R1, L     // Saut au label L
                // si R1 différent de 0
  code(expression)
L:...
```

- L'instruction CMP est semblable à l'instruction SUB, si ce n'est que quand SUB fait un dépassement de capacité, CMP produit un résultat dont la valeur est fausse mais le signe est correct.
- Le saut au label L doit être transformé en un déplacement entre l'instruction de saut et la cible du saut.

Production des sauts dans le compilateur

- Pour produire un saut, on dispose de deux méthodes :

```
int pc()
  // retourne le pointeur de programme courant

void fixup(int from, int to)
  // stocke (to-from) en tant qu'entier signé
  // de 16 bits dans le champ de l'opérande c
  // à l'adresse from.
```


Production des sauts dans le compilateur (2)

- Pour faire un saut en avant au label L, on procède ainsi :

```
int jumpadr = code.pc();
// produit un saut avec 0 dans l'opérande c
// produit les autres instructions jusqu'au label L
code.fixup(jumpadr, code.pc());
```

- tandis que pour faire un saut en arrière, on procède ainsi :

```
int jumptarget = code.pc();
// produit les instructions jusqu'au saut
int jumpadr = code.pc();
// produit un saut avec 0 dans l'opérande c
code.fixup(jumpadr, jumptarget);
```

Items pour les conditions

- Comment représenter les conditions sous forme d'item ?
- On doit pouvoir effectuer un saut en fonction de la valeur de la condition.
- Donc, on doit stocker dans l'item :
  - le numéro du registre dans lequel le résultat de la condition est stocké,
  - le code de l'instruction de saut à utiliser si la condition est fausse.
- le constructeur d'un tel item se présente donc ainsi :

```
class CondItem extends Item {
  RegisterItem reg;
  int jmpcode;
  CondItem(RegisterItem reg, int jmpcode) {
     this.reg = reg;
     this.jmpcode = jmpcode;
  }
```


Interface des items pour les conditions

- Quelles opérations doivent être fournies par les items conditionnels ?
- On peut charger une valeur booléenne dans un registre :

```
var b: Int = (x == y);
```

- Ils doivent donc offrir la même interface que les autres :

```
RegisterItem load();
```

- Ils peuvent aussi être utilisés pour contrôler un saut conditionnel. Ils doivent donc aussi offrir la méthode :

```
int jumpIfFalse();
   // produit un saut qui est pris quand la condition
   // est fausse, avec 0 comme opérande c.
   // Retourne l'adresse de l'instruction de saut.
```

- Il est parfois nécessaire d'inverser une condition :

```
CondItem negate();
   // inverse le code de l'instruction de saut
   // pour inverser la condition du saut.
```

Production des CondItems

- Les CondItems peuvent apparaître dans deux situations :
  - Comme résultat d'une opération de comparaison, p.ex.

```
x == y, x <= y
```

  - Dans ce cas, on construit un CondItem directement, p.ex. pour `x <= y`:

```
RegisterItem xi = // résultat du chargement de x
RegisterItem yi = // résultat du chargement de y
emit (CMP, xi.register, xi.register, yi.register)
Code.freeRegister(yi);
result = new CondItem (xi, BGT);
```

- Les CondItems peuvent aussi être construits à partir de valeurs booléennes. P.ex :

```
if (done) printInt(sum) else ()
if (head(myList)) ...
```


Production des CondItems (2)

- On doit donc pouvoir produire un CondItem à partir d'un item quelconque.
- Dans ce but, on ajoute la méthode `makeCondItem` à la classe Item.

```
class Item {
  ...
  /** Produit un CondItem qui effectue un saut
   *  si la valeur de cet item est fausse.
   */
  CondItem makeCondItem () {
    return new CondItem (load(), BEQ);
  }
```

- Pour les `CondItem`s, cette méthode ne fait rien :

```
class CondItem extends Item {
   ...
   CondItem makeCondItem () { return this; }
}
```

Production de code pour les conditions simples

- Pour produire le code pour une condition `x OP y` où `OP` est parmi `==, !=, <, >, <=, >=`, on procède ainsi :
  - Chargement des opérandes `x, y` dans des items de registres, p.ex. `xi` et `yi`.
  - Production d'une comparaison CMP entre `xi` et `yi`:

```
code.emit(CMP, xi.register, xi.register, yi.register);
code.freeRegister(yi);
```

  - Production d'un CondItem:

```
result = new CondItem(xi, opcode(operator));
```

- Ici, `opcode(operator)` retourne l'opcode à utiliser pour un saut si la condition est fausse en fonction de l'opérateur. P.ex.

```
==          BNE
!=          BEQ
<           BGE
>           BLE
<=          BGT
>=          BLT
```


Production de code pour les énoncés conditionnels

- On peut maintenant expliquer comment les énoncés conditionnels simples (if-then) seraient traduits (s'ils existaient en misc) :

Grammaire abstraite : E ::= Cond E1 E2

Code à produire :

```
        code(E1)
        BF L          // pseudo code pour: « saut si faux »
        code(E2)
L:      ...
```

Schéma de production :

```
CondItem c = generate(E1).makeCondItem ();
int jumpAdr = c.jumpIfFalse();
generate(E2);
code.fixup(jumpAdr, Code.pc());
```

Production de code pour les énoncés conditionnels (2)

- On procède de manière similaire pour les énoncés conditionnels doubles (if-then-else) : S ::= Cond E1 E2 E3

Code à produire :

```
        code(E1)
        BF L1          // pseudo code pour: « saut si faux »
        code(E2)
        BRA L2         // pseudo code pour: « saut toujours »
L1:     code(E3)
L2:     ...
```

Schéma de production :

```
CondItem c = generate(E1).makeCondItem();
int jumpAdr1 = c.jumpIfFalse();
generate(E2);
int jumpAdr2 = code.pc();
emit(BEQ, 0, 0);
code.fixup(jumpAdr1, code.pc());
generate(E3);
code.fixup(jumpAdr2, code.pc());
```


Mise en œuvre de la classe CondItem

- Pour résumer, les CondItem doivent offrir :

```
class CondItem extends Item {
    RegisterItem reg;
    int jmpcode;
    CondItem(RegisterItem reg, int jmpcode) {
        this.reg = reg:
        this.jmpcode = jmpcode;
    }
    RegisterItem load() { ... }
    int jumpIfFalse() { ... }
    CondItem negate() { ... }
    CondItem makeCondItem() { ... }
}
```

Mise en œuvre de *load*

- Idée : traiter

```
var b: Int = (x <= y);
```

comme :

```
var b: Int = (if (x <= y) true else false);
```

Où vrai (*true*) est représenté par 1 et faux (*false*) est représenté par 0.

- On doit donc écrire :

```
RegisterItem load() {
  int jumpAdr1 = this.jumpIfFalse();
  code.emit(ADDI, reg.register, 0, 1);
  int jumpAdr2 = code.pc();
  code.emit(BEQ, 0, 0);
  code.fixup(jumpAdr1, code.pc());
  code.emit(ADDI, reg.register, 0, 0);
  code.fixup(jumpAdr2, code.pc());
}
```


Evaluation « court-circuitée »

- Rappel :
  En misc, les opérateurs « et » et « ou » logique sont du sucre syntaxique.
- `A & B` est traduit comme `if (A) B else false`
- `A | B` est traduit comme `if (A) true else B`
- Cette traduction permet de ne pas évaluer `B` lorsque cela n'est pas strictement nécessaire.
- On nomme cette technique *évaluation « court-circuitée »* (*short circuit evaluation* en anglais).
- Ce type d'évaluation n'est pas obligatoire, c'est un choix fait lors de la conception du langage.
- Java offre ce choix au programmeur : les opérateurs doubles (`&&` et `||`) court-circuitent, les simples (`&` et `|`) pas.

Exemple de code

La traduction de

```
if (x == 0 & y > 0) 12 else 13
```

est équivalente à celle de :

```
if (if (x == 0) y > 0 else false) 12 else 13
```

à savoir:

```
    LDW   1,SP,#x
    BNE   1,L3      // x == 0 ?
    LDW   1,SP,#y
    BLE   1,L1      // y > 0
    ADDI  1,0,1     // true (y > 0)
    BRA   L2
L1: ADDI  1,0,0     // false (y <= 0)
L2: BRA   L4
L3: ADDI  1,0,0     // false
L4: BEQ   1,L5
    ADDI  1,0,12
    BRA   L6
L5: ADDI  1,0,13
L6: ...
```


Comment éviter les sauts et chargements inutiles

- Le code de l'exemple précédent n'est pas optimal, étant donné qu'il affecte 0 ou 1 à R1 dans le seul but d'effectuer ensuite un saut en fonction de cette valeur.
- Il serait mieux d'effectuer le saut directement et de ne jamais produire la valeur 0/1.
- On obtient ainsi du meilleur code :

```
    LDW   1,SP,#x
    BNE   1,L1      // x == 0 ?
    LDW   1,SP,#y
    BLE   1,L1      // y > 0
    ADDI  1,0,12
    BRA   L2
L1: ADDI  1,0,13
L2: ...
```

- Comment peut-on produire un tel code ? Pas possible en gardant le sucre syntaxique...

Schéma d'évaluation « court-circuité »

- Pour obtenir du code plus concis, on saute directement depuis les endroits intermédiaires d'une condition à la cible finale (c-à-d qu'on utilise des courts-circuits).
- Problème : une condition peut désormais contenir plusieurs sauts qui doivent tous être corrigés.
- On a deux sorte de sauts : sauts si vrai, sauts si faux.
- Idée : On garde l'ensemble de tous les sauts qui pointent vers un même label au moyen d'une liste chaînée de type *Chain* :

```
class Chain {
  int jumpAdr;
  Chain next;
}
```

- Quand on atteint le label cible d'une chaîne, on corrige toutes les adresses de saut qu'elle contient.

Schéma d'évaluation « court-circuité » (2)

- Les items conditionnels sont désormais caractérisés par quatre éléments :
  - L'instruction « saut si faux » finale et le registre d'après lequel sauter.
  - Une chaîne pour tous les sauts qui sont effectués car on sait que la condition est vraie.
  - Une chaîne pour tous les sauts qui sont effectués car on sait que la condition est fausse.
- Les deux opérations suivantes sur les chaînes sont utiles :

```
/** corrige tous les sauts de la chaîne c pour
qu'ils pointent sur target */
void patch (Chain c, int target) { ... }
/** Forme l'union de deux chaînes */
Chain union (Chain c1, Chain c2) { ... }
```

CondItems pour l'évaluation court-circuitée

```
Class CondItem extends Item {
     RegisterItem reg;
     int jumpCode;
     Chain trueJumps;
     Chain falseJumps;
     CondItem(RegisterItem reg, int jumpCode, Chain trueJumps, Chain
falseJumps) {
        ... Comme d'habitude ...
     }
     void load() {
       // semblable à avant, mais on doit corriger à la fois
       // trueJumps et falseJumps
       ...
       patch (trueJumps, code.pc());
       emit (ADDI, 1, 0, 1);
       int jumpadr = code.pc();
       emit (BEQ, 0, 0);
       patch (falseJumps, code.pc());
       emit (ADDI, 1, 0, 0);
       code.fixup(jumpadr, code.pc());
     }
```


CondItems pour l'évaluation court-circuitée

```
     CondItem makeCondItem() { return this; }

     /** Produit l'instruction pour sauter lorsque la condition
       * est fausse. Retourne la chaîne des « sauts si faux » */
     Chain jumpIfFalse() {
       return new Chain (code.emit(jumpCode,reg.register,0),
                         falseJumps);
     }

     /** Produit l'instruction pour sauter lorsque la condition
       * est vraie. Retourne la chaîne des « sauts si vrai » */
     Chain jumpIfTrue() {
       return negate().jumpIfFalse();
     }

     CondItem negate() {
       return new CondItem(reg,jumpCode^1,falseJumps,trueJumps);
     }

   }
```

Production de code avec courts-circuits pour les expressions booléennes

- On s'intéresse uniquement à quelques exemples représentatifs :

E = Operation Lt E_1 E_2

```
r1 = generate(E1).load();
r2 = generate(E2).load();
freeRegister(r2);
code.emit(CMP, r1, r1, r2);
return new CondItem(r1, BGE, null, null);
```

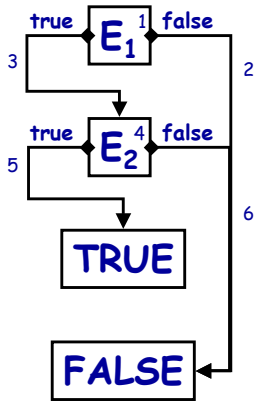
| Operation Not E

```
return generate(E)
  .makeCondItem()
  .negate();
```


Production de code avec courts-circuits pour les expressions booléennes (2)

| Operation And E_1 E_2

```
1  Item lcond = generate(E1).makeCondItem();
2  Chain falseJumps = lcond.jumpIfFalse();
3  patch (lcond.trueJumps, code.pc())
4  Item rcond = generate(E2).makeCondItem();
   return new CondItem(
     rcond.jumpCode,
5    rcond.trueJumps,
6    union(falseJumps, rcond.falseJumps));
```



| Operation Or E_1 E_2 ?

Production de code avec court-circuits pour les énoncés

- On examine uniquement l'énoncé conditionnel (if-then-else); les autres sont semblables et un peu plus simples.

S = Cond E E_1 E_2

```
Item cond = generate(E).makeCondItem();
Chain falseJumps = cond.jumpIfFalse();
patch (cond.trueJumps, code.pc());
generate(E1);
int elseJump = code.emit(BRA);
patch (falseJumps, code.pc());
generate(E2);
code.fixup(elseJump, code.pc());
```


Résumé

- On a vu comment le compilateur traduit les conditions et les énoncés de contrôle en séquences de conditions et d'instructions de saut.
- Trois schémas existent pour la traduction d'opérateurs booléens :
  - Traduction en opérations logiques (pas très bon étant donné que la seconde opérande est toujours évaluée).
  - Traduction en if-then-else (simple, mais inefficace) .
  - Traduction avec court-circuits (plus efficace).
- La complexité de la production de code est principalement cachée dans la notion d'item.