Partie VIII : Production de code II

- Contexte de compilation et visiteurs
- Organisation du générateur de code
- Visiteur pour les énoncés
- Visiteur pour les expressions
- Allocation de registres
- Visiteur pour les conditions
- Labels
- Gestion des opérateurs logiques

Contexte de compilation

- Lors de la production de code, il est souvent nécessaire de connaître le *contexte* dans lequel on opère.
- Exemple : lors de la compilation d'une *expression* comme `1+x*y`, il faut savoir dans quel registre placer le résultat de l'évaluation de l'expression.
- Autre exemple : lors de la compilation d'une *condition* comme `x<y`, il faut savoir où sauter si la condition est vraie/fausse.
- Dans certains cas, le contexte n'est pas important. Exemple : production de code pour une définition de fonction.

Contextes et visiteurs

- Comment intégrer les contextes au visiteur de production de code ?
- Étant donné que le contexte va du haut de l'arbre vers le bas, il faut le passer en *argument* au visiteur.
- Exemple : visiteur pour la production de code des expressions (contexte : registre destination).

```
class Generator implements Visitor {
  private int targetReg;
  public void generate(Tree tree, int targetReg) {
    int bkpTargetReg = this.targetReg;
    this.targetReg = targetReg;
    tree.apply(this);
    this.targetReg = bkpTargetReg;
  }
  public void caseUnitLit(UnitLit tree) {
    emit(ADDI, targetReg, 0, 0);
  }
  // et ainsi de suite...
}
```


Contextes et visiteurs (2)

- Comment intégrer les autres contextes, p.ex. celui nécessaire à la production de code pour les conditions ?
- Première idée : on ajoute ce contexte au visiteur précédent.
- Problème : le visiteur prend en argument l'union de tous les contextes possibles, mais un seul est valide à un moment donné.
- Meilleure solution : *plusieurs visiteurs*, un par type de contexte.
- Pour le projet, nous avons identifié trois contextes :
  1. le contexte utilisé pour la compilation d'énoncés (vide),
  2. le contexte utilisé pour la compilation d'expressions,
  3. le contexte utilisé pour la compilation de conditions.
- Nous définissons donc trois visiteurs.

Organisation du générateur de code

```
class Generator {
  private Gen genVisitor = new Gen();
  private GenLoad genLoadVisitor = new GenLoadVisitor();
  private GenCond genCondVisitor = new GenCondVisitor();

  public void gen(Tree tree) { genVisitor.generate(tree); }
  public void genLoad(Tree tree, int targetReg) {
    genLoadVisitor.generate(tree, targetReg);
  }
  // idem pour genCond.

  private class Gen implements Visitor {
    public void generate(Tree tree) { tree.apply(this); }
    public void caseProgram(Program tree) { ... }
    // et ainsi de suite...
  }
  private class GenLoad implements Visitor {
    public void generate(Tree tree, int targetReg) { ... }
    public void caseUnitLit(UnitLit tree) { ... }
    // et ainsi de suite ...
  }
  private class GenCond implements Visitor { ... }
}
```

instances

méthodes

visiteurs

Visiteur avec méthode par défaut

- Petit problème : comme chacun des visiteurs ne traite qu'un sous-ensemble des nœuds, beaucoup de cas sont identiques.
- Exemple : le visiteur `Gen`, dont le contexte est vide, ne gère pas directement les nœuds `If`, `Assign`, `Binop`, `Ident`, etc. qui sont traités par `GenLoad`. Le code pour tous ces nœuds, dans `Gen`, est donc strictement identique.
- Comment gérer cela dans le cadre des visiteurs ?
- Avec un nouveau type de visiteur, comportant un cas par défaut !

Visiteur avec méthode par défaut (2)

```
abstract public class DefaultVisitor implements Visitor {
  public void caseProgram(Program tree) { caseDefault(tree); }
  public void caseBody(Body tree)       { caseDefault(tree); }
  public void caseFunDef(FunDef tree)   { caseDefault(tree); }
  public void caseFormal(Formal tree)   { caseDefault(tree); }
  // ... et ainsi de suite pour tous les cas
  public abstract void caseDefault(Tree tree);
}

class Generator {
  // ... comme avant
  private class Gen extends DefaultVisitor {
    public void generate(Tree tree) { tree.apply(this); }
    public void caseProgram(Program tree) { ... }
    // et ainsi de suite...
    public void caseDefault(Tree tree) { ... }
  }
  private class GenLoad extends DefaultVisitor {
    // ...
  }
  // idem pour GenCond
}
```


Visiteur pour les définitions et énoncés

- Le visiteur `Gen` gère tous les nœuds qui ne produisent pas de valeur, à savoir :
  - les définitions (`Program`, `FunDef`, `Formal`, `VarDef`),
  - les énoncés (`While`, `Exec`).
- Son contexte est donc vide : `void gen(Tree tree)`
- Que faire pour les autres nœuds (cas par défaut) ?
- Exemple de code misc : `while (true) 5+2;`
- Ici, l'expression `5+2` est utilisée comme si elle ne produisait pas de valeur. De manière générale, n'importe quelle expression peut être utilisée de la sorte en misc.
- Solution : dans le cas par défaut, on alloue un registre temporaire, dans lequel on charge la valeur de l'expression, puis on libère le registre sans utiliser la valeur qu'il contient.

Visiteur pour les expressions

- Le visiteur `GenLoad` gère tous les nœuds qui produisent une valeur, à savoir :
  - les opérateurs arithmétiques (`+`, `-`, `*`, `/`, `%`),
  - les opérateurs de construction et de décomposition de listes (`::`, `head`, `tail`),
  - les constantes (entiers, liste vide, valeur « unit »),
  - les autres expressions : application de fonction, conditionnelle (`if`), bloc, affectation, identificateur.
- Son contexte est composé du registre dans lequel la valeur finale doit être placée :

```
void genLoad(Tree tree, int targetReg)
```


Visiteur pour les expressions (2)

- Que faire pour les autres nœuds (cas par défaut) ?
- Exemple de code misc : `var v: Int = (1 < 2);`
- Ici, la condition `1<2` est utilisée comme une expression, c-à-d qu'on calcule sa valeur sous forme d'un booléen, au lieu de sauter quelque part en fonction du résultat du test.
- De manière générale, toute condition peut être utilisée comme une expression, en misc.
- Solution : on enrobe les conditions utilisées comme expressions dans un **`if`**, de manière à contourner le problème.
- Le code ci-dessus devient alors :

```
var v: Int = if (1 < 2) 1 else 0;
```

- Nouvel invariant : toute condition apparaît dans la partie « condition » d'un **`if`**.

Exemple de cas pour genLoad

- Examinons un cas simple pour genLoad : les constantes entières.

```
void caseIntLit(IntLit tree) {
  int highBits = tree.value >> 16;
  int lowBits  = tree.value & 0xFFFF;
  if (highBits == 0) {
    code.emit(ORIU, targetReg, 0, lowBits);
  } else {

    ???

  }
}
```


Allocation de registres

- La classe Code contient des méthodes pour gérer les registres.
- Elle gère les registres non pas comme une pile mais comme un ensemble, ce qui est plus général sans être plus compliqué.
- Interface

```
// Alloue et retourne un registre, lève une exception s'il
// n'y en a plus.
int getRegister();

// Alloue le registre donné, lève une exc. s'il est alloué
int getRegister(int reg);

// Libère le registre donné, lève une exc. s'il est libre.
void freeRegister(int reg);

// Retourne l'ensemble des registres actuellement alloués.
// Si la position i du tableau est vraie, alors Ri est alloué.
boolean[] usedRegisters();
```

Allocation des registres (2)

```
private boolean[] freeRegisters = new boolean[32];

public Code() {
  for (int r = RC_MIN; r <= RC_MAX; ++r) freeRegisters[r] = true;
}

public int getRegister() {
  for (int r = RC_MIN; r <= RC_MAX; ++r)
    if (freeRegisters[r]) return getRegister(r);
  throw new Error("no more free registers");
}

public int getRegister(int reg) {
  if (!freeRegisters[reg])
    throw new Error("register already allocated");
  freeRegisters[reg] = false;
  return reg;
}

public void freeRegister(int reg) {
  if (freeRegisters[reg])
    throw new Error("register already free");
  if (reg < RC_MIN || reg > RC_MAX)
    throw new Error("attempt to free special register");
  freeRegisters[reg] = true;
}
```


Allocation de registres et genLoad

- Invariant : le registre destination (`targetReg`) est alloué.
- Il peut donc être utilisé comme registre temporaire au besoin.
- Exemple : pour produire le code pour l'addition, on peut passer directement ce registre à la fonction de production de code pour le sous-arbre gauche.

```
void casePlus(Plus tree) {
  genLoad(tree.left, targetReg);
  int rightReg = code.getRegister();
  genLoad(tree.right, rightReg);
  code.emit(ADD, targetReg, targetReg, rightReg);
  code.freeRegister(rightReg);
}
```

- Avantage : on réduit le nombre de registres utilisés.

Allocation de registres et genLoad (2)

- Autre petit truc concernant `targetReg` : on peut le libérer temporairement, s'il peut être utile à d'autres.
- Attention : il faut toujours être certain de pouvoir le réallouer au moment où on en aura besoin !
- Exemple : production de code pour les blocs (nœud `Body`) :

```
void caseBody(Body tree) {
  ...
  code.freeRegister(targetReg);
  gen(tree.stats);
  code.getRegister(targetReg);
  genLoad(tree.expr, targetReg);
  ...
}
```

peut utiliser `targetReg` au besoin

- Permis par l'invariant : les méthodes de production (`gen`, `genLoad` et `genCond`) libèrent *tous* les registres qu'elles ont alloué avant de retourner.

Visiteur pour les conditions

- Le visiteur `GenCond` gère tous les nœuds qui testent une condition, à savoir :
  - les opérateurs de comparaison (<, <=, ==, !=, >=, >),
  - la primitive de test sur les listes, `isEmpty`,
  - (si ce n'était pas du sucre syntaxique) les opérateurs logiques (&, |, !).
- Que contient son contexte ?
- Première idée : où sauter quand la condition est vraie, et où sauter quand la condition est fausse.
- Mais un des deux est toujours la prochaine instruction.
- Conclusion : le contexte contient *un seul* endroit où sauter, et un booléen spécifiant *quand* sauter : si la condition est vraie, ou si elle est fausse.

```
void genCond(Tree t, Label targetLabel, boolean when)
```

Visiteur pour les conditions (2)

- Que faire pour les nœuds qui ne sont pas des conditions (cas par défaut) ?
- Exemple de code misc : `if (f(5)) 12 else 14`
- Ici `f(5)` est un appel de fonction, pas directement une condition.
- De manière générale, n'importe quelle expression dont la valeur est de type entier (rigoureusement : booléen) peut être utilisée comme condition en misc.
- Solution : charger la valeur de l'expression, au moyen de `genLoad`, puis tester si sa valeur est fausse, c.-à-d. nulle.

Labels

- Comment représenter l'endroit où sauter ?
- En assembleur DLX : déplacement par rapport à l'instruction de saut.
- Déplacements peu pratiques à manipuler, et en cas de saut vers l'avant, *inconnus* au moment de la production du saut !
- Abstraction : labels.
- Un label représente une *position* dans le code. Il peut être dans deux états :
  - libre, si sa position effective n'est pas encore connue,
  - ancré, dans le cas contraire.
- Au moment où un label est créé, il est libre.
- Lorsqu'il est finalement ancré, les instructions de saut qui pointent vers lui sont automatiquement corrigées.

Labels (2)

```
public class Label {
  List toPatch = new LinkedList();   // Instructions à corriger
  int pc = -1;                       // Position du label

  boolean isAnchored() { return pc >= 0; }

  int getAnchor() { return pc; }

  void setAnchor(int pc) {
    this.pc = pc;
    Iterator toPatchIt = toPatch.iterator();
    while (toPatchIt.hasNext()) {
      int instrPC = ((Integer)toPatchIt.next()).intValue();
      fixup(instrPC, pc);
    }
  }

  void recordInstructionToPatch(int pc) {
    toPatch.add(new Integer(pc));
  }
}
```

corrige l'instruction à l'adresse instrPc pour qu'elle saute à pc

Labels (3)

- La classe Code contient des méthodes d'émission d'instructions qui acceptent directement des labels.

```
void emit(int opcode, int a, Label l) {
  if (l.isAnchored())
    emit(opcode, a, (l.getAnchor() - pc()) / WORD_SIZE);
  else {
    l.recordInstructionToPatch(pc());
    emit(opcode, a, 0, 0);
  }
}
```

- Elle contient également des méthodes pour créer et ancrer des labels.

```
Label getLabel() { return new Label(); }

void anchorLabel(Label l) { l.setAnchor(pc()); }
```

Labels (4)

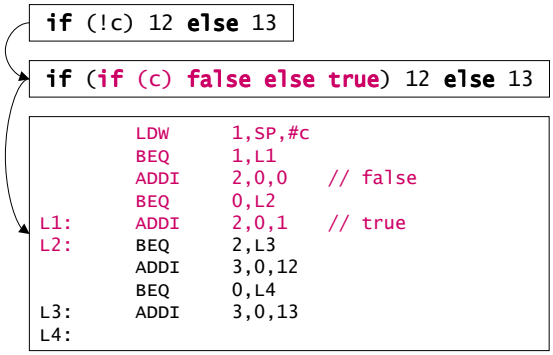
- La production de code utilisant des labels devient très simple.
- Exemple : nœud `if`

```
void caseIf(If tree) {
  Label elseLabel = code.getLabel();
  genCond(tree.cond, elseLabel, false);
  genLoad(tree.thenp, targetReg);
  Label afterLabel = code.getLabel();
  code.emit(BEQ, 0, afterLabel);
  code.anchorLabel(elseLabel);
  // et ensuite ?


}
```


Opérateurs logiques

- En misc, les opérateurs logiques (&, | et !) sont traités comme du sucre syntaxique.
- Solution simple, mais produisant du relativement mauvais code.
- Exemple :



Opérateurs logiques (2)

- Comment faire mieux ? Simple : supprimer le sucre syntaxique.

```
void caseNot(Not tree) {
  genCond(tree.expr, targetLabel, !when)
}
```

- On obtient ainsi du bien meilleur code, car aucune instruction n'est produite pour la négation logique !

```
if (!c) 12 else 13
```

```
        LDW     1,SP,#c
        BNE     1,L1
        ADDI    3,0,12
        BEQ     0,L2
L1:     ADDI    3,0,13
L2:
```


Opérateurs logiques (3)

- Autre exemple : conjonction (« et » logique)

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

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

```
        LDW     1,SP,#x
        BNE     1,L3    // x == 0 ?
        LDW     1,SP,#y
        BLE     1,L1    // y > 0
        ADDI    1,0,1   // true (c-à-d y > 0)
        BRA     L2
L1:     ADDI    1,0,0   // false (c-à-d 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:
```

Opérateurs logiques (4)

Pour produire le meilleur code suivant :

```
        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:
```

renoncer une fois de plus au sucre syntaxique :

```
void caseAnd(And tree) {
  if (when) {
    Label afterLabel = code.getLabel();
    genCond(tree.left, afterLabel, false);
    genCond(tree.right, targetLabel, true);
    code.anchorLabel(afterLabel);
  } else {
    genCond(tree.left, targetLabel, false);
    genCond(tree.right, targetLabel, false);
  }
}
```


Opérateurs logiques (5)

- Dernier opérateur : disjonction (« ou » logique)
- Idée : le traiter comme du sucre syntaxique grâce aux lois de De Morgan

```
a | b  <=>  !(!a & !b)
```

- Question : le code ainsi produit est-il bon ?

Résumé

- La production de code dépend souvent du contexte.
- Pour le compilateur misc, trois contextes identifiés :
  - le contexte pour les définitions et énoncés (vide),
  - le contexte pour les expressions (registre destination),
  - le contexte pour les conditions (label destination, condition de saut).
- L'identification des contextes est empirique et dépend des langages source (ici misc) et cible (ici assembleur DLX).
- Organisation du générateur de code :
  - un visiteur par contexte,
  - visiteurs mutuellement récursifs.
- Abstraction utile à la production de code : labels.