Production de code et gestion de la mémoire

Martin Odersky

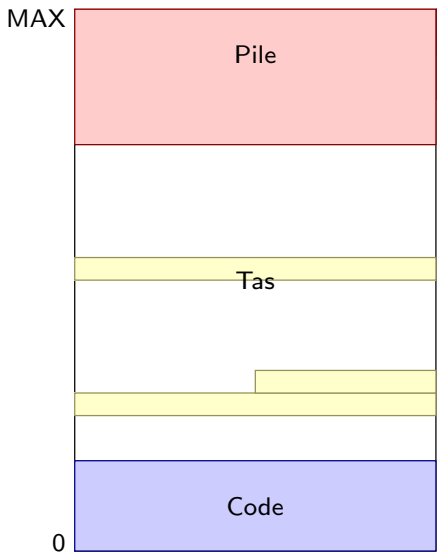
16 et 17 janvier 2006
version 1.1

Plan du cours

Organisation de la mémoire

En Zwei, on utilise la mémoire pour trois raisons :

1. Stockage du code,
   au bas de la mémoire, à partir de l'adresse 0.
2. Stockage des variables locales et des paramètres,
   sur la pile, qui croît vers le bas depuis le sommet de la mémoire.
3. Stockage des structures de données dynamiques (objets),
   sur le tas, entre le code et la pile.



Allocation et libération de mémoire

La quasi-totalité des langages de programmation actuels permettent l'allocation dynamique de mémoire.

- En C, on utilise la fonction `malloc` de la bibliothèque standard.
- En C++, Java, Scala ou Zwei, on utilise l'opérateur `new`.

Si on a à disposition une quantité infinie de mémoire, il n'est jamais nécessaire de libérer la mémoire allouée devenue inutile.

En pratique, la mémoire disponible est toujours limitée. Il existe deux manière de la libérer :

1. Explicitement (`free` en C, `delete` en C++), une manière très dangereuse qui est une des sources principales de bogues dans les logiciels.
2. Automatiquement, au moyen d'un glaneur de cellules ou ramasse-miettes (*garbage collector*), comme en Lisp, Java, Scala, Zwei, etc., qui simplifie la vie du programmeur en libérant automatiquement la mémoire qui n'est plus accessible depuis le programme.

Le tas

On appelle tas (*heap*) la zone de la mémoire dans laquelle la mémoire dynamique est allouée.

- Le tas est géré par le système de gestion de la mémoire, qui garde une liste des emplacements libres et alloués dans le tas.
- Le système de gestion de la mémoire offre des services d'allocation et de libération de mémoire dynamique. La libération peut être automatique.

Gestion de la mémoire dans DLX

Le simulateur DLX que nous vous fournissons possède un système de gestion de la mémoire basé sur un glaneur de cellules.

- Peu réaliste, mais bien pratique pour le projet.
- Allocation mémoire au moyen d'une instruction `SYSCALL`.

Exemple : Allocation d'un bloc de mémoire

On alloue un bloc de 25 octets sur le tas, l'adresse de premier octet étant placée dans `R2`

```
ADDI    R1 R0 25
SYSCALL R2 R1 SYS_GC_ALLOC
```

- Il est inutile de libérer la mémoire allouée : le système s'en charge.

Représentation des objets Zwei

Un objet en Zwei est composé de deux parties :

1. la valeur de ses champs, spécifique à l'instance,
2. ses méthodes, partagées par toutes les instances de la classe.

Les données d'instance (valeur des champs) sont représentées par un blocs de mémoire alloué dynamiquement par l'opérateur `new` sur le tas.

- Ce bloc mémoire contient en plus une adresse pointant vers les données de classe,
- plus de détail à ce sujet dans un cours ultérieur.

Pour manipuler les objets (passage en argument, stockage dans des variables, etc.) on utilise leur adresse.

- On dit qu'on manipule les objets par référence (ou par pointeur), comme le fait Java ou Scala. En C/C++ les objets peuvent aussi être manipulés par valeur.
- La constante `null` est représentée par l'adresse 0, qui ne fait jamais partie du tas, car elle contient la première instruction du programme.

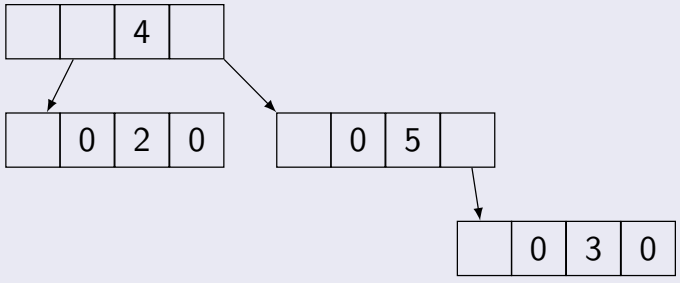
Notez que les objets Zwei sont immuables, il n'est pas possible de modifier leur contenu.

Exemple : Représentation des objets Zwei

Le code Zwei suivant :

```
class Tree { Tree left; int value; Tree right; }
new Tree(new Tree(null, 2, null),
         4,
         new Tree(null,
                  5,
                  new Tree(null, 3, null)))
```

produit la situation suivante en mémoire :



Contextes de compilation

Lors de la production de code, il est souvent nécessaire de connaître le contexte dans lequel on opère.

- 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.
- Lors de la compilation d'une condition comme $x < y$, il faut savoir où sauter si la condition est vraie ou fausse.

Dans d'autres cas, le contexte n'est pas important.

- Lors de la production de code pour une définition de fonction.

Contextes et décomposition fonctionnelle

Comment intégrer les contextes au visiteur de production de code ?

- Etant donné que le contexte va du haut de l'arbre vers le bas, il faut le passer en argument à la fonction de génération.

Exemple : Générateur de code pour les expressions

Le contexte dans ce cas est le registre de destination.

```
class Generator(analyzer: Analyzer) {
  def genLoad(tree: Expr, targetReg: Int): Unit =
    tree match {
      case IntLit(value) =>
        emit(ADDI, targetReg, 0, value)
      ...
    }
}
```

Comment intégrer les autres contextes, comme celui nécessaire à la production de code pour les conditions ?

- Première idée : on ajoute ce contexte à la méthode de génération précédente.
  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 méthodes, une par type de contexte.

Pour le projet, nous avons identifié trois contextes :

1. le contexte utilisé pour la génération d'énoncés (qui est vide),
2. le contexte utilisé pour la génération d'expressions,
3. le contexte utilisé pour la génération de conditions.

Nous définissons donc trois méthodes de génération.

Petit problème : comme chacune des méthodes de génération ne traite qu'un sous-ensemble des noeuds, beaucoup de cas sont identiques.

- La méthode de génération d'énoncés, dont le contexte est vide, ne gère pas les noeuds `If`, `Binop`, `Ident`, etc. qui produisent une erreur.
- Le code pour tous ces noeuds est donc strictement identique dans cette méthode.

En Scala, il suffit d'utiliser le cas par défaut du filtrage de motifs !

Les définitions et énoncés

La méthode de génération des énoncés gère tous les noeuds qui ne produisent pas de valeur, à savoir :

- les définitions (`Program`, `ClassDef`, `FieldDecl`, `MethodDef`),
- les énoncés (`While`, `Var`, `Set`, `Do`, `PrintInt`, `PrintChar`).

Son contexte est vide :

```
def gen(tree: Tree): Unit = {
  ...
}
```


Les expressions

La méthode de génération des expressions gère tous les noeuds qui produisent une valeur, à savoir :

- les opérateurs arithmétiques ($+$, $-$, $*$, $/$, $\%$),
- l'opérateur `new`,
- la sélection de champ ou de méthode,
- les constantes entières et `null`,
- les applications de fonction, expressions conditionnelles (`If`), bloc, identificateur.

Son contexte est composé du registre dans lequel la valeur finale doit être placée :

```
def genLoad(tree: Expr, targetReg: Int): Unit = {
  ...
}
```

Les expressions : les conditions

Dans le code `int v = (1 < 2)`, la condition `1 < 2` est utilisée comme une expression :

- 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 :
Il suffit d'enrober les conditions utilisées comme expressions dans un `if`, de manière à contourner le problème.

- Le code ci-dessus devient alors
  `int v = if (1 < 2) 1 else 0`.

Nouvel invariant : une condition apparaît seulement dans la partie "condition" d'un `if` ou d'un `while`.

Les expressions : charger une constante

Exercice

Considérons le cas de `genLoad` pour les constantes :

```
case IntLit(value) =>
  val highBits = value >> 16;
  val lowBits = value & 0xFFFF;
  if (highBits != 0) {
    // TODO
  } else
    code.emit(ORIU, reg, ZERO, lowBits);
```

Allocation des 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.

Elle fournit les méthodes suivantes :

```
class Code {
  // Alloue et retourne un registre, leve une exception s'il
  // n'y en a plus.
  def getRegister(): Int;
  // Alloue le registre donne, leve une exc. s'il est alloue
  def getRegister(r: Int): Int;
  // Libere le registre donne, leve une exc. s'il est libre.
  def freeRegister(r: Int): Unit;
  // Retourne l'ensemble des registres actuellement alloues.
  // Si la position i du tableau est vraie, alors Ri est alloue.
  def usedRegisters(): Array[Boolean];
}
```


Voici le code de la classe `Code` :

```
class Code {
  val ZERO = 0; val RC_MIN = 1; val RC_MAX = 29;
  val FP = 30; val LINK = 31;
  private val used = new Array[Boolean](32);
  used(ZERO) = true; used(FP) = true; used(LINK) = true;
  def getRegister(): Int = {
    var i = RC_MIN;
    while (i <= RC_MAX && used(i)) i = i + 1;
    if (i <= RC_MAX) i else throw new Error("no␣more␣free␣registers")
  }
  def getRegister(r: Int): Int = {
    if (used(r)) throw new Error("getting␣non-free␣register␣R" + r);
    used(r) = true;
    r
  }
  def freeRegister(r: Int): Unit = {
    if (!used(r) || r < RC_MIN || r > RC_MAX)
      throw new Error("freeing␣already-free␣or␣special␣register␣R" + r);
    used(r) = false
  }
  def usedRegisters(): Array[Boolean] = used.clone()
}
```

Utiliser le registre de destination

On sait que le `targetReg` est alloué dans `genLoad`.

Il peut donc être utilisé comme registre temporaire, réduisant le nombre de registres utilisés.

Exemple : Génération de l'addition

On peut passer `targetReg` à la méthode de génération pour le sous-arbre gauche :

```
case Binop(op, left, right) => op match {
  case Operators.ADD =>
    genLoad(left, targetReg);
    val rightReg = code.getRegister();
    genLoad(right, rightReg);
    code.emit(opcode(op), targetReg, targetReg, rightReg);
    code.freeRegister(rightReg);
  case _ => ...
}
```


Autre petit truc concernant `targetReg` : on peut le libérer temporairement, s'il peut être utile ailleurs.

Attention, il faut toujours être certain de pouvoir le réallouer au moment où on en aura besoin !

Exemple : Génération du bloc

```
case Block(stats, main) =>
  ...
  code.freeRegister(targetReg);
  stats foreach gen;
  code.getRegister(targetReg);
  genLoad(main, targetReg);
  ...
```

Ceci est permis grâce à l'invariant suivant :

Les méthodes de génération (`gen`, `genLoad` et `genCond`) libèrent tous les registres qu'elles ont alloués avant de retourner.

Les conditions

La méthode de génération des conditions gère tous les noeuds qui testent une condition, à savoir :

- les opérateurs de comparaison (`<`, `<=`, `==`, `!=`, `>=`, `>`),
- les opérateurs logiques (`&&`, `||`, `!`).

Que doit contenir son contexte ?

- On imagine : où sauter quand la condition est vraie, et où sauter quand la condition est fausse.
- Mais une de ces deux destinations est toujours l'instruction suivante sur l'architecture DLX (et les processeurs modernes).

Son contexte contient l'endroit où sauter, et quand sauter :

```
def genCond(tree: Expr,
            targetLabel: code.Label,
            when: Boolean): Unit = {...}
```


Les conditions — conditions illogiques

Problème : Dans le code `if (f(5)) 12 else 14`, la condition de saut `f(5)` est un appel de fonction, pas une condition.

De manière générale, n'importe quelle expression dont la valeur est de type entier (rigoureusement : booléen) peut être utilisée comme condition en Zwei.

Solution : charger la valeur de l'expression, au moyen de `genLoad`, puis tester si sa valeur est vraie, c'est-à-dire différente de `0`.

Le code ci-dessus est ainsi compilé comme
`if (f(5) != 0) 12 else 14`.

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 !

On utilise une abstraction : les labels.

- Un label représente une position dans le code. Il peut être dans deux états :
  1. libre, si sa position effective n'est pas encore connue,
  2. ancré (*anchored*), 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.

Les labels dans le compilateur Zwei sont des instances de `Label` :

```
class Label {
  private var toPatch = List[Int]();
  private var pc = -1;
  def setAnchor(pc: Int): Unit = {
    this.pc = pc;
    toPatch foreach { instrPC =>
      // corrige l'instruction a l'adresse instrPC pour
      // qu'elle saute a pc
      code((pc - instrPC) / WORD_SIZE - 1).c =
        (pc - instrPC) / WORD_SIZE; }
    }
  def getAnchor(): Int = pc;
  def isAnchored(): Boolean = pc >= 0;
  def recordInstructionToPatch(pc: Int): Unit =
    toPatch = pc :: toPatch;
}
```

La classe `Code` contient des méthodes d'émission d'instructions qui acceptent directement des labels :

```
def emit(opcode: Int, a: Int, l: Label): Unit =
  if (l.isAnchored())
    emit(opcode, a, (l.getAnchor() - pc()) / WORD_SIZE);
  else {
    l.recordInstructionToPatch(pc());
    emit(opcode, a, Integer.MIN_VALUE, Integer.MIN_VALUE);
  }
```

Elle contient également des méthodes pour créer et ancrer des labels :

```
def getLabel(): Label = new Label();
def anchorLabel(l: Label): Unit = l.setAnchor(pc());
```


La production de code utilisant des labels devient très simple.

Exemple : Génération du noeud `if`

```
case If(cond, thenp, elsep) =>
  val elseLabel = code.getLabel();
  genCond(cond, elseLabel, false);
  genLoad(thenp, targetReg);
  val afterLabel = code.getLabel();
  code.emit(BEQ, ZERO, afterLabel);
  code.anchorLabel(elseLabel);
  ...
```

Opérateurs logiques

Comment compile-t-on la négation logique ?

```
case Unop(op, expr) =>
  if (op == Operators.NOT)
    genCond(expr, targetLabel, _____);
  else ...
```

En d'autres termes, la négation logique ne produit aucun code, mais inverse simplement une instruction.

Exemple : Génération d'une négation logique

Le code `if (!c) 12 else 13` devient :

```
    LDW  R1 SP #c
    BNE  R1 L1
    ADDI R2 0  12
    BEQ  0  L2
L1: ADDI R2 0  13
L2:
```


Qu'en est-il de la disjonction ?

```
case Binop(op, left, right) => op match {
  case Operators.AND =>
    if (when) {
      val afterLabel = code.getLabel();
      genCond(left, __________, _____);
      genCond(right, __________, _____);
      code.anchorLabel(afterLabel);
    } else {
      genCond(left, __________, _____);
      genCond(right, __________, _____)
    }
  ...
}
```

Le code obtenu est bon, car il court-circuite l'évaluation (*short-circuit evaluation*) d'une condition dès que sa valeur est connue.

En C, C++, Java et Scala l'évaluation de la disjonction est court-circuitée. Connaissez-vous un langage pour lequel cela n'est pas le cas ?

Dernier opérateur : la conjonction ("et" logique).

On le traite comme du sucre syntaxique grâce aux lois de De Morgan : `a & b` $\Longleftrightarrow$ `!(!a | !b)`.

Exercice : Performance du sucre

Le code ainsi produit est-il bon, ou pourrait-on faire mieux en gérant directement la conjonction ?

Contextes de compilation — Résumé

La production de code dépend souvent du contexte.

Pour le compilateur Zwei, trois contextes identifiés :

1. pour les définitions et énoncés (vide),
2. pour les expressions (registre destination),
3. pour les conditions (label destination, quand sauter).

L'identification des contextes est empirique et dépend des langages source (Zwei) et cible (assembleur DLX).

Le générateur de code est organisé avec :

- une méthode de génération par contexte,
- des méthodes de génération mutuellement récursives.

Les labels sont une abstraction utile à la production de code.

Visiteurs pour Java

La génération de code implique une décomposition fonctionnelle (visiteur) sur l'arbre plus complexe que précédemment :

- on utilise le cas par défaut du filtrage de motifs,
- ainsi que plusieurs visiteurs mutuellement récursifs (`gen`, `genLoad` et `genCond`).

En Java, où l'on utilise le motif de conception "visiteur", il faut :

- utiliser un nouveau type de visiteur qui supporte les cas par défaut et
- définir, dans le visiteur principal, une nouvelle classe à instance unique (*singleton*) pour chaque type de visiteur.

Visiteurs avec valeur par défaut

Un nouveau type de visiteurs qui supporte les cas par défaut :

- déclare une méthode abstraite `caseDefault` que les sous-classes doivent implanter,
- déclare toutes les méthodes `caseXYZ` comme un appel à la méthode `caseDefault`.

Ainsi, les méthodes `caseXYZ` qui n'auront pas été redéfinies dans la classe d'implantation exécuteront l'opération par défaut :

```
abstract public class DefaultVisitor implements Visitor {
  public void caseUnop(Unop tree) { caseDefault(tree); }
  public void caseCall(Call tree) { caseDefault(tree); }
  ...
  public abstract void caseDefault(Tree tree);
}
```

Exemple : Visiteur GenLoad avec cas par défaut

```
private class GenLoad extends DefaultVisitor {
  public void generate(Tree tree, int targetReg) {
    ...
    tree.apply(this);
    ...
  }
  public void caseUnop(Unop tree) {...}
  public void caseCall(Call tree) {...}
  public void caseDefault(Tree tree) {...}
}
```


Visiteurs multiples dans Generator

```
public class Generator implements RISC {
  // instances singleton des visiteurs
  private final Gen genVisitor = new Gen();
  private final GenLoad genLoadVisitor = new GenLoad();
  private final GenCond genCondVisitor = new GenCond();
  // methodes de visite
  public void gen(Tree tree) { genVisitor.generate(tree); }
  private void genLoad(Tree tree, int targetReg) {
    genLoadVisitor.generate(tree, targetReg);
  }
  private void genCond ...
  // classes des visiteurs
  private class Gen extends DefaultVisitor {
    public void generate(Tree tree) { tree.apply(this); }
    ...
  }
  private class GenLoad extends DefaultVisitor {...}
  private class GenCond extends DefaultVisitor {...}
  ...
}
```