Production de code
gestion de la mémoire et contextes

Martin Odersky

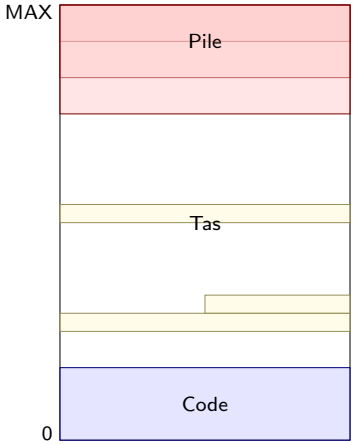
version 1.5

Plan du cours

Organisation de la mémoire

Typiquement, la mémoire est utilisée 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 Drei, on utilise l'opérateur `new`.

Si l'on dispose d'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, Drei, 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 Drei

Un objet en Drei 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.

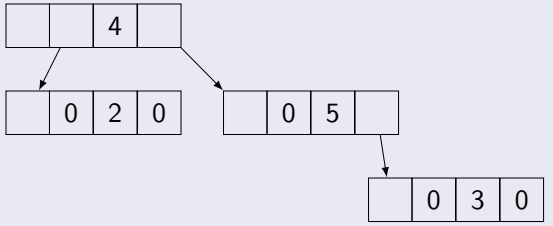
Notez que les objets Drei sont immuables, il n'est pas possible de modifier leur contenu.

Exemple : Représentation des objets Drei

Le code suivant (en Drei, sauf pour le `null`, représenté par l'adresse 0) :

```
class Tree { val left: Tree; val value: Int;
             val right: Tree; }
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

On distingue trois modes de générations, différenciés par leur conventions et leurs besoins.

- *Expressions valeurs* comme `1+x*y`.
  Chaque expression résulte à l'exécution en une valeur stockée dans un registre. Par convention, le résultat est placé dans le registre du sommet de la pile des registres.
- *Expressions conditionelles* comme `0 < x && x <= 10`.
  Elles deviennent des séquences de sauts conditionnels. Pour chaque condition, on doit savoir s'il faut sauter quand elle est vraie ou fausse, et où sauter.
- *Enoncés* comme `var x = y + 1`.
  Le code des énoncés n'est pas dépendant d'autres éléments, si ce n'est par l'ordre.

Méthodes de génération de code

Chaque mode de génération correspond à une méthode.

`genLoad`	pour les expressions valeurs.
`genCond(targetLabel, when)`	pour les conditions
`gen`	pour les énoncés

Les définitions et énoncés

La méthode gen 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`, `If`, `Var`, `Set`, `Do`, `PrintInt`, `PrintChar`).

Son contexte est vide :

```
def gen(tree: Tree): Unit
```


Les expressions

La méthode genLoad 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 applications de fonction, bloc, identificateur.

Elle ne prend pas d'arguments, mais, par contrat, elle doit retourner avec un registre frais alloué au sommet de la pile et contenant la valeur de l'expression :

```
def genLoad(tree: Expr): Unit
```


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
```


Question : génération des conditions

Considérant

```
if (0 <= x && x < 10)
  y = x
else
  y = 10
```

Quel code assembleur faut-il générer.

Labels

Comment représenter l'endroit où sauter ?

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.
- Les instructions qui réfèrent un label libre réfèrent une *position potentielle* dans l'avenir du programme.
- Lorsque le label est finalement ancré, les positions *potentielles* des instructions précédentes peuvent être concrétisées.

La gestion des labels dépend du langage vers lequel on compile.

Pour un assembleur qui supporte lui-même des labels (comme DLX) :

- on imprime toutes les instructions avec le nom du label,
- l'ancrage du label imprime son nom à l'endroit voulu.

Pour du code binaire :

- quand un label libre est utilisé dans une instruction, on mémorise la position de cette instruction,
- quand le label est ancré (et que son adresse physique est connue), on visite les instructions qui y réfèrent, et l'on met à jour l'adresse,
- quand un label ancré est utilisé dans une instruction, on imprime cette fois directement l'adresse.

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

```
class Label(name: String) extends Instruction {
  def setAnchor: Unit = code += this
  override def toString() = name
  def write(out: PrintWriter): Unit = {
    out.print(name)
    out.println(":")
  }
}
```

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

```
def emit(opcode: Int, a: Int, l: Label): Unit
```


Structures de contrôle : if

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)
  gen(thenp)
  val afterLabel = code.getLabel
  code.emit(BEQ, ZERO, afterLabel)
  elseLabel.setAnchor
  gen(elsep)
  afterLabel.setAnchor
```

Structures de contrôle : while et repeat

Question : Génération du noeud while

Quel code assembleur faut-il générer pour
`while (x >= 0) x = x / 2 - 1` ?

Exemple : Génération du noeud while

```
case While(cond, body) =>
  val startLabel = code.getLabel
  startLabel.setAnchor
  val endLabel = code.getLabel
  genCond(cond, endLabel, false)
  gen(body)
  emit(BEQ, 0, startLabel)
  endLabel.setAnchor
```


Comparaisons

On compile les comparaisons vers des sauts conditionnels. L'instruction `cmp` permet d'obtenir le signe de la comparaison. Le saut se définit sur ce résultat et la valeur du paramètre `when`.

Exemple : Génération des comparaisons

```
def genCond(cond: Tree,
            targetLabel: Label,
            when: boolean) =  ...
case Binop(LT, left, right) =>
  genLoad(left)
  genLoad(right)
  emit(CMP, sndReg, topReg, topReg)
  dropReg
  emit(if (when) BLT else BGE, topReg, targetLabel)
  ...
```


Disjonction logique (OR)

En C, C++, Java et Scala l'évaluation de la disjonction est court-circuitée. Dans quel langage n'est ce pas le cas ?

La disjonction est générée différemment si le saut doit avoir lieu quand la condition est vraie ou fausse.

Exemple : génération des disjonctions

```
case Binop(OR, left, right) =>
  if (when) {
    genCond(left, targetLabel, true)
    genCond(right, targetLabel, true)
  } else {
    val afterLabel = code.getLabel
    genCond(left, afterLabel, true)
    genCond(right, targetLabel, false)
    afterLabel.setAnchor }
```

Conjonction logique (AND)

La génération du code de la conjonction est opposée à celle de la disjonction.

Exercice : génération des conjonctions

Quel schéma de génération utilisera-t-on pour les conjonctions ?

```
case Binop(OR, left, right) =>
  if (when) {
    ...
  } else {
    ...
  }
```


Négation

La génération du code des négations est très simple.

```
case Unop(NOT, expr) =>
  genCond(expr, targetLabel, !when)
```

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 > 0) 12 else 13` devient :

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


Conjonctions : suite

En Drei, le OR est traité 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 ?

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 Drei.

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`.

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 prédicat dans une expression `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`.

Contextes de compilation — Résumé

La production de code dépend souvent du contexte.

Pour le compilateur Drei, trois contextes identifiés :

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

L'identification des contextes est empirique et dépend des langages source (Drei) 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) {
    ...
    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) {
    genLoadVisitor.generate(tree);
  }
  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 {...}
  ...
}
```