Introduction à la génération de code pour Deca sans objet

version : 18 avril 2023 à 13:47

L'objectif de cette feuille est de vous faire mettre en place *incrémentalement* des algorithmes de génération de code pour la [**MachineAbstraite**] qui respecte la [**Sémantique**] du Deca sans objet. Il faut essayer ici se limiter à des algorithmes simples, qui seront *généralisables*, au fur et mesure des incréments, *y compris* quand on considérera l'extension *orientée objet* de Deca. Inutile donc de chercher à optimiser le code assembleur généré (puisque ce n'est pas l'objectif du projet). Il faut mieux appliquer des principes *compositionnels* de compilation. Typiquement, on cherche à compiler chaque *instruction* indépendamment les unes des autres. La compilation des *expressions* est imbriquée dans celle des instructions, mais là aussi, il faut essayer d'avoir des principes très compositionnels, en anticipant sur les futurs incréments.

En particulier, dès l'incrément "print-variable", on met en place des conventions de représentation des variables qui sont compatibles avec [**ConventionsLiaison**]. Ce document décrit en effet ce qu'on appelle traditionnellement l'Application Binary Interface (ABI), c'est-à-dire comment on encode les *procédures* et les variables/paramètres au niveau du langage machine.

Pour commencer, les registres R0 et R1 sont des registres écrasables (*scratch*) [1] par les procédures appelées : typiquement R0 sert de paramètre de retour dans les appels de procédures, et R1 sert de paramètre pour les instructions assembleurs d'entrée/sortie. Les autres registres – à partir de R2 – sont les registres "*non-volatiles*" ou "*callee-saved*", et leur valeur doit être préservées/restaurées par les procédures appelées. Pour faciliter la compatibilité future avec ces conventions, au niveau "print-variable", on place le résultat des expressions dans le registre R2. [2]

1 Variables et autres expressions atomiques (litéraux, etc)

On représente chaque variable d'un bloc de code donné par une position de la pile. Par exemple, la première variable est à l'adresse 1(LB), la deuxième à 2(LB), etc. [3]

Exercice 1 (Début de l'incrément "print-variable"). On considère le programme ci-dessous. Pour la génération de code, on alloue `x` dans 1(LB) et `y` dans 2(LB). Appliquer les principes ci-dessus pour générer (à la main) le code assembleur de ce programme. Ici, vous ne devez utiliser que deux registres : R1 comme paramètre à WINT ou WFLOAT, et R2 pour stocker le résultat (temporaire) des expressions. NB : vérifiez que votre programme assembleur fonctionne avec l'interpréteur IMA.

```
{
  int x;
  float y;
  x = 17;
  y = 0.125;
  println(x, ";", y);
}
```

Exercice 2 (Compilation de `readInt`). Pour compiler `readInt`, il va falloir gérer une étiquette pour traiter le cas d'erreur. Sinon le résultat de `readInt` sera typiquement copié dans `R2` avant d'être traité par l'instruction englobante. Appliquez ce principe sur le programme suivant où `x` est alloué dans 1(LB).

1. On parle aussi de registres "*volatiles*" ou "*caller-saved*".
2. Plus tard, les sous-expressions pourront mettre leur résultats dans les registres de numéro supérieur à R2, ce qui garantit qu'un résultat dans une sous-expression n'est jamais écrasé par les appels de procédures qui peuvent arriver dans les autres sous-expressions.
3. Dans le bloc principal, on a LB=GB, donc on peut aussi utiliser 1(GB), 2(GB), etc. La version avec LB est aussi directement applicable aux blocs des méthodes.

```
{
  int x = readInt();
  print(x,":",readInt());
}
```

2 Expressions booléennes

Pour traiter les comparaisons, on applique le schéma général des expressions booléennes (i.e. les "conditions" au sens de la règle 3.29 de [**SyntaxeContextuelle**]). Ce schéma est expliqué section 7.2 de [**Gencode**]. On l'illustre ici sur le code "`x <= 421`" où `x` est alloué dans 1(LB). Il y a deux cas :

— le cas $\langle Code(\texttt{x <= 421}, vrai, E)\rangle$ qui doit soit se brancher à l'étiquette E si `x <= 421`, soit continuer en séquence sinon. Du coup, ce code est simplement :

```
LOAD 1(LB), R2
CMP #421, R2
BLE E
```

— le cas $\langle Code(\texttt{x <= 421}, faux, E)\rangle$ qui doit soit continuer en séquence si `x <= 421`, soit se brancher à l'étiquette E sinon. Ici, le code est identique au précédent, sauf qu'on prend le code condition complémentaire :

```
LOAD 1(LB), R2
CMP #421, R2
BGT E
```

Exercice 3 (If-then-else avec comparaisons). Appliquer ce principe pour compiler le programme suivant, vu dans le TD d'introduction à Deca. Il faudra aussi utiliser un schéma de compilation des if-then-else comme celui proposé en section 8.1 de [**Gencode**] (vous avez le droit d'adapter/simplifier le schéma proposé à cet endroit).

```
{
  int x ;
  print("Entrez un entier:");
  x = readInt();
  if (17 < x) {
    println("17 < ", x);
  } else {
    println(x, " <= 17");
  }
}
```

Exercice 4 (Opérateurs logiques). Étendez le principe aux opérateurs booléens – comme indiqué dans la section 7.2 de [**Gencode**] – sur l'exemple suivant :

```
{
  int x = readInt();
  print("Votre nombre ");
  if (! ((7 <= x && x < 13) || (17 < x && x <= 42))) {
    print("n'est pas");
  } else {
    print("est");
  }
  println(" dans [7,13[ union ]17,42]");
}
```

3 Expressions arithmétiques

Trouver un algorithme de génération de code pour les expressions arithmétiques qui va bien s'adapter à l'extension objet de Deca n'est pas complètement évident. Il faut prendre en compte que des appels de méthodes arbitraires peuvent avoir lieu dans une sous-expression : le code généré pour ces appels de méthode doit néanmoins respecter [**ConventionsLiaison**]. Notamment, les appels de méthodes devront préserver les valeurs des registres autres que R0 et R1. Ces autres registres peuvent donc servir à sauvegarder les résultats des sous-expressions.

Par exemple, le code assembleur de "`(x1*x2)-((x3*x4)+(x5*x6))`" peut commencer par utiliser R2 pour stocker le résultat de "`x1*x2`" puis R3 pour stocker le résultat de "`x3*x4`", puis R4 pour stocker le résultat de "`x5*x6`", puis faire un "`ADD R4, R3`", puis "`SUB R3, R2`".

Il faut aussi noter ici que la génération de code des expressions arithmétiques doit respecter l'ordre d'évaluation gauche-droite (cf. section 4 de [**Semantique**]).

Enfin, il faut considérer qu'il n'y a qu'un nombre limité de registres et que lorsque tous les registres contiennent déjà un résultat intermédiaire, il faudra sauvegarder certains résultats intermédiaires dans la pile. Ici on appelle RMAX le registre maximal autorisé où MAX vaut `X-1` si on est en train de compiler avec `decac -r X` (cf. [**Decac**]), ou 15 sinon. [4]

Notre algorithme va donc ici utiliser indirectement le registre `SP` (via `PUSH` et `POP`). Il faudra donc *systématiquement* bien mettre-à-jour `SP` à l'entrée du bloc *pour prendre en compte* la réservation des variables sur la pile. Autrement dit, un bloc avec n variables doit commencer par un "`ADDSP #`n".

Ci-dessous, on formalise cet algorithme par la fonction récursive <codeExp(e,n)> qui génère le code assembleur calculant le résultat d'une expression arithmétique e dans le registre Rn où n est supposé dans 2..MAX, et en n'utilisant comme registres que R1 et les registres de Rn à RMAX. Ici, e est manipulée sous la forme d'un *arbre abstrait* du langage **EXPR**, que pour simplifier, on suppose limité à :

EXPR ::= `Identifier`↑*Symbol*
| `IntLiteral`↑*int*
| `Plus`[**EXPR EXPR**]
| `Minus`[**EXPR EXPR**]
| `Mult`[**EXPR EXPR**]
| `Assign`[`Identifier`↑*Symbol* **EXPR**]
| `ReadInt`

L'algorithme suppose aussi que chaque identificateur *symb* est associé à une opérande `dval` (au sens de [**MachineAbstraite**]) de la forme n(LB) ou n(GB). On note ici `@`*symb* cette opérande.

Avec la description de <codeExp(e,n)> précédente, on peut déjà donner les cas du `Assign` et du `ReadInt`.

```
<codeExp( ReadInt , n)>
  := RINT
     BOV io_error
     LOAD R1, Rn

<codeExp( Assign[ Identifier↑symb  e] , n)>
  := <codeExp( e , n)> // calcul de e dans Rn (avec n ∈ 2..MAX)
     STORE Rn, @symb
```

Pour les autres cas, on définit ensuite une fonction <dval(e)> qui retourne soit une `dval` si e est un identificateur ou un littéral, soit $\perp$ (i.e. le pointeur nul) sinon.

4. L'option `-r` de Deca permet de débogguer plus facilement la gestion des registres dans votre implémentation.

```
<dval(IntLiteral↑n)> := #n
<dval(Identifier↑symb)> := @symb
<dval(op[ _ _ ])> := ⊥ pour op ∈ {Plus, Minus, Mult, Assign}
<dval(ReadInt)> := ⊥
```

Ainsi, le cas des identificateurs et des littéraux peut être traité de manière uniforme par :

```
<codeExp(e, n)> avec <dval(e)>≠⊥
  := LOAD <dval(e)>, Rn
```

On définit aussi la fonction <mnemo(*op*)> qui retourne l'opérateur IMA associé à un opérateur arithmétique.

```
<mnemo(Plus)> := ADD
<mnemo(Minus)> := SUB
<mnemo(Mult)> := MUL
```

Cela permet de traiter le cas des opérateurs arithmétiques. L'algorithme distingue trois sous-cas donnés à la figure 1.

```
<codeExp(op[ e1  e2 ] ,n)> avec <dval(e2)>≠⊥
  := <codeExp(e1 ,n)>
     <mnemo(op)> <dval(e2)>, Rn

<codeExp(op[ e1  e2 ] ,n)> avec <dval(e2)>=⊥ et n < MAX
  := <codeExp(e1 ,n)>
     <codeExp(e2 ,n+1)>
     <mnemo(op)> Rn+1, Rn

<codeExp(op[ e1  e2] ,n)> avec <dval(e2)>=⊥ et n = MAX
  := <codeExp(e1 ,n)>
     PUSH Rn ; sauvegarde
     <codeExp(e2 ,n)>
     LOAD Rn, R1
     POP Rn  ; restauration
     <mnemo(op)> R1, Rn
```

FIGURE 1 – `codeExp` des opérateurs arithmétiques, avec $op \in \{\texttt{Plus}, \texttt{Minus}, \texttt{Mult}\}$.

Exercice 5 (Expressions arithmétiques). Appliquez le principe ci-dessus pour traiter l'exemple suivant avec MAX=15.

```
{
  int x;
  x = readInt();
  x = ((x - 7) + readInt()) * 5;
  print("resultat:", x);
}
```

Exercice 6 (Expressions arithmétiques avec limitation de registres). Appliquez le principe ci-dessus pour traiter l'exemple suivant avec MAX=3 (i.e. pour `decac -r 4`).

```
{
  int x1, x2;
  x1=readInt();
  x2=readInt();
  print("resultat:", x1 - (7 + (x2 * 5)));
}
```

Pour incorporer les expressions arithmétiques dans les expressions booléennes, il faut considérer que les comparaisons fonctionnent comme un opérateur arithmétique de mnémonique `CMP`, qui est suivi d'un branchement conditionnel juste après. Autrement dit, $\langle Code(e1 \texttt{ <= } e2, vrai, E)\rangle$ donne schématiquement le code ci-dessous :

<codeExp(**CMP** [$e1$ $e2$], 2)> ; **CMP** *comme mnémonique IMA de la racine*
BLE E

Exercice 7 (Expressions booléennes avec arithmétique). Appliquez ça pour traiter l'exemple suivant (avec MAX=15).

```
{
  int x1 = readInt(), x2 = readInt();
  if (3 * x1 <= 7 * x2) {
    println("Gagné !");
  } else {
    println("Perdu !");
  }
}
```