Compilation

Génération de code assembleur MIPS

Jules Chouquet

SOM2IF15 – 2024

Résultat de l'étape précédente

Traduction de l'AST en représentation intermédiaire

- En gros, le résultat est : `List<Pair<Frame, List<Command>>>`
- Pour chaque fonction, on a :
  - Un `Frame`, qui traduit l'en-tête de la fonction
  - Une `List<ir.com.Command>` qui est une traduction du corps de la fonction.
- La syntaxe des commandes de la représentation intermédiaire est proche de celle des instructions assembleur MIPS.
- Mais certaines de ces commandes contiennent des **expressions**
- Il n'y a pas d'expressions en assembleur.

Rappel : Représentation intermédiaire

Expressions

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & number \\
& | & \texttt{reg}_i \\
& | & unop\ \langle expr \rangle \\
& | & \langle expr \rangle\ binop\ \langle expr \rangle
\end{array}
$$

Génération de code MIPS

Principes

- Generation d'une `List<String>`

Expressions

- Compilation à l'aide d'une machine à pile :
- Pour chaque (sous)-expression :
  - calcul de la valeur
  - empilement de la valeur
- Pour une expression binaire : calculer les deux sous-expressions, dépiler deux valeurs, et empiler le résultat

Taille des valeurs et opérations de pile

```
static int sizeOf(ir.Type type) {
    if (type == Type.BYTE) return 1;
    return 4;
}
static List<String> push(String register) {
    List<String> asmCode = MakeList.one(command("sub␣$sp,␣4"));
    asmCode.add(command("sw␣" + register + ",␣4($sp)"));
    return asmCode;
}

static List<String> pop(String register) {
    List<String> asmCode =
  MakeList.one(command("lw␣" + register + ",␣4($sp)"));
    asmCode.add(command("add␣$sp,␣4"));
    return asmCode;
}
```

Constantes

```
@Override
public List<String> visit(Byte exp) {
    List<String> asmCode =
  MakeList.one(Asm.command("li␣$t0,␣" + exp.getValue()));
    asmCode.addAll(Asm.push("$t0"));
    return asmCode;
}

@Override
public List<String> visit(Int exp) {
    List<String> asmCode =
  MakeList.one(Asm.command("li␣$t0,␣" + exp.getValue()));
    asmCode.addAll(Asm.push("$t0"));
    return asmCode;
}
```

Registres temporaires

Approche naïve

- On les stocke sur la pile
- Un registre = un décalage par rapport au cadre d'activation courant.

Rappel

- la pile augmente par adresses décroissantes
- `$fp` début du cadre courant
- `$sp` fin du cadre courant
- on utilise la pile pour stocker les résultats intermédiaires : `$sp` change tout le temps
- `$fp` change seulement lors des appels de procédures

$\Longrightarrow$ les registres seront associés à un décalage par rapport à `$fp`

Exemple : Code et traduction intermédiaire

```
int select
  (bool theFirst,
   int first,
   int second)
{
  if (theFirst)
    return first
  else
    return second
}
```

```
==== FRAME: L0:
  FRAME INFO:
    {entryPoint=L0: , exitPoint=L1:
     parameters=[reg0, reg1, reg2],
     result=Optional[reg3],
     locals=[], size=0}
  CODE: [CJump (reg0, L2: , L3: ),
          L2: ,
              reg3 := reg1,
              goto L1: ,
          L3: ,
              reg3 := reg2,
              goto L1: ]
==== END FRAME
```

Exemple : cadre d'activation

- Tous les registres temporaires stockés sur la pile
- Dans l'exemple : 4 registres temporaires
- Pour simplifier : sur la pile toutes les valeurs seront d'une taille égale à celle des mots en machine (32 bits dans notre cas)
- Sur le tas, les valeurs auront leur vraie taille.

Génération de code MIPS pour les expressions

IR Expressions to MIPS

- Constantes & expressinos binaires : déjà définies

Lecture de registre

- `offset` dans une association regAlloc : `Map<Register, Integer>`
- ce décalage est par rapport à `$fp`
- la taille par défaut est 32 bits.
- lecture : `lw $t0, offset($fp)`
- le résultat est poussé sur la pile.

Rappel : commandes de la représentation intermédiaire

Instructions

$\langle com \rangle$	::=	*label*:	Étiquette
	|	$\texttt{reg}_i := \langle expr \rangle$	écriture dans un registre
	|	$\texttt{reg}_i + \langle expr \rangle := \langle expr \rangle$	(tableaux le cas échéant)
	|	`jump` $(\langle expr \rangle)$ *label*, *label*	saut conditionnel
	|	`goto` *label*	saut inconditionnel
	|	`call` *frame* $\langle expr \rangle, \ldots, \langle expr \rangle$	appel (fonc/syst)
	|	$\texttt{reg}_i :=$ *frame* $\langle expr \rangle, \ldots, \langle expr \rangle$	appel (fonc/syst)

Génération de code pour les instructions

- `goto` *label* : `j label`
- écriture dans un registre :
  - compilation de l'expression : l'exécution du code généré pousse la valeur sur la pile.
  - registre : récupérer son décalage (`offset`) depuis une association (register $\rightarrow$ offset).
  - code final : $code_{\text{exp}} pop(\texttt{\$t0})[\texttt{save \$t0}, offset(\texttt{\$fp})]$
    où `save` est :
    - ★ `sw` (si le type est `INT` ou `ADDRESS` (tableaux))
    - ★ `sb` (si le type est `BYTE`)
- Écrire dans la mémoire (tableaux) : comme pour les registres, mais :
  - On doit connaître l'adresse du tableau (elle est dans un registre)
  - On doit calculer le décalage pour une case donnée
  - L'indice est dans la première expression : le code généré le pousse sur la pile.
  - L'indice assembleur est $4 + sz \times i$ où sz est la taille d'une case.

Génération de code pour les appels

- Première étape : passage de paramètres (Appel-Par-Valeur uniquement).
  Les quatre premiers paramètres dans $a0...$a3. (simplification possible : interdire les fonctions à plus de 4 paramètres lors de l'analyse sémantique. Sinon, utiliser la pile.)
- Deuxième étape : appel de la fonction.
- Les autres apspects de la fonction seront traités lors de la génération de code pour son `Frame` (au tout début de la génération), et les *fragments* de code associés (à la fin de la génération).

Frames et Fragments

- Un fragment est une liste d'instructions
- C'est la concaténation des des `List<String>` obtenues pour chaque instruction du corps.
- Les `Frame` représentent les fonctions.
- D'abord :
  - le code pour sauvegarder les registres nécessaires (ceux qui doivent être préservés par l'appelé en MIPS)
  - le code pour mettre à jour `$fp` et `$sp`
  - le code pour associer les registres MIPS pour les paramètres et les registres de la représentation intermédiaire.
- Le corps de la fonction (le fragment évoqué plus haut).
- Enfin :
  - le code pour restituer les registres (préservés par l'appelant)
  - si c'est une fonction (avec une valeur à retourner) : copier le résultat dans `$v0`
  - sauter à l'appelant

Generation de code pour le programme

Le bout du chemin

- Les programmes en MIPS doivent contenir des directives `.text` (et éventuellement `data`).
- Ils doivent comporter une étiquette `main`
- Un appel système explicite est nécessaire à la sortie.
- La génération de code pour chaque cadre d'activation.

Est-ce qu'on a vraiment fini ?

Oui, mais...

La génération de code est très naïve

- Un peu mieux : utiliser les registres du processeur autant que possible pour traduire les registres de la représentation intermédiaire, et utiliser la pile quand il n'y a plus de registres (voir TP5)
- Encore mieux : faire une analyse (statique) du code intermédaire pour déterminer quels registres temporaires sont utilisés en même temps (c'est *l'analyse de vie*), et utiliser un algorithme de coloration de graphe pour associer les registres temporaires *vivants* aux registres du processeur.

Sélection d'instructions

Une optimisation classique est de modifier la suite d'instructions du code intermédiaire pour que la génération de code assembleur soit optimale (certaines instructions peuvent être rassemblées)

Fin

Bonus : quelques indications pour les tableaux.

Tableaux et chaînes de caractères

- La valeur d'un tableau est son adresse
- En supposant que la taille de chaque case est sz qu'il y a n cases :
  - 4 octets au début pour stocker le nombre des cases (n)
  - $sz \times n$ octets pour les valeurs des cellules
- Pour une chaîne comme `"hello"` :
  - nombre de cases : nombre de caractères + 1
  - en mémoire :
    - ★ 4 octets pour la taille : valeur `int`= 6
    - ★ 6×1 octets pour les caractères `'h'`, `'e'`, `'l'`, `'l'`,`'o'`, `'\0'`

Génération de code – lecture mémoire : $\texttt{reg}_i[\langle expr \rangle]$

- L'adresse d'un tableau dans $\texttt{reg}_i$: obtenue par une lecture de registre, pas besoin de le pousser sur la pile, supposons qu'elle est dans `$t0`.
- Le décalage est $\langle expr \rangle$: on le compile récursivement, et le résultat (`int`) sera stocké sur la pile.
- On dépile l'indice : supposons qu'il est dans `$t1`
- Si i est le décalage, sz la taille des cellules, en MIPS : $4 + sz \times n$. Il faut générer le code pour calculer cette valeur. sz est obtenue avec `sizeOf`, le type du registre.
  Supposons que le code stocke cet index MIPS dans `$t2`
- Selon la taille : `lb` ou `lw`
- `load $t3, $t2($t0)` est impossible : le décalage doit être un littéral.
- À la place : ajouter `$t0` à `$t2` et utiliser `load $t3, ($t2)`
- Et enfin, empiler `$t3`.