Projet de compilation (Étape 5)

Samuel Tardieu
SE202 - Année scolaire 2016/2017

Cinquième étape

Avant de commencer

Avant de commencer à travailler, il vous faut intégrer le code venant de l'équipe pédagogique. Depuis votre dépôt :

```
% git pull template step5
```

À ce stade, votre dépôt a été enrichi de l'étape `step4` du dépôt de référence. Vérifiez que les tests de base fonctionnent toujours :

```
% workon se202
(se202)% python -m unittest
```

La première commande fait appel à votre environnement de développement virtuel `tiger`, si ce n'était pas déjà fait. La seconde cherche tous les tests unitaires du dépôt et les exécute.

Travail à faire

Dans cette étape, nous allons générer du code pour un processeur ARM. Le nombre de registres dont nous disposerons pour l'instant est infini, l'allocateur de registre sera fait lors d'une étape ultérieure.

Méthode

Lors de la génération de code, nous allons parcourir la définition de chaque fonction en représentation intermédiaire (IR) et générer les instructions ARM correspondant. Pour cela, il faut pour chaque nœud trouver l'instruction assembleur couvrant au mieux la fonctionnalité du nœud et descendre dans l'arbre si c'est nécessaire.

Exemple

```
MOVE(MEM(BINOP("+", TEMP("sp"), -4)),
     TEMP("fp"))
```

peut par exemple se décomposer ainsi en passant par un registre intermédiaire :

```
add temp0, sp, #-4
str fp, [temp0]
```

ou, et on cherchera à favoriser cette situation,

```
str fp, [sp, #-4]
```

Encapsulation

Nous allons commencer à générer du texte (de l'assembleur) qui correspond à la cible visée. Par contre, nous avons toujours besoin de conserver des informations sur les registres manipulés et les sauts pour pouvoir les manipuler par la suite.

Nous allons encapsuler les instructions assembleur dans des objets dérivant de la classe `Instr` (dans `codegen/instr.py`).

La classe `Instr`

Les classes dérivant de `Instr` implémentent toutes plusieurs méthodes :

- `defs()` donne la liste des `Temp` dont le contenu est défini par cette instruction.
- `uses()` donne la liste des `Temp` dont le contenu est utilisé par cette instruction.
- `jumps()` donne la liste des labels auxquels cette instruction peut sauter (une liste vide indique qu'elle passe à l'instruction suivante).

Affichage de l'instruction

Les instances disposent d'un champ `template` dans lequel se trouve la modèle de l'instruction, à qui on passe une liste composée de `defs + uses` en utilisant la méthode `format` des chaînes en Python.

Par exemple, une opération *load* chargeant le contenu de la mémoire pointé par `r0` dans `r1` pourra utiliser comme template `"str {}, [{}]"`. L'affichage se fera donc en appelant

```
"ldr {}, [{}]".format("r0", "r1")
```

soit `ldr r0, [r1]`.

Permutation de l'ordre des arguments

Il est parfois nécessaire d'utiliser les arguments dans un ordre différent de l'ordre donné (d'abord les `defs` puis les `uses`). On pourra utiliser l'ensemble des spécificateurs de `format()`:

```
"str {1}, [{0}]".format("r4", "r5")
```

donnera `str r5, [r4]`.

À quoi servent ces méthodes ?

Les méthodes `defs()` et `uses()` permettront par la suite de calculer quels sont les registres vivants lors de l'exécution de cette instruction.

La phase d'allocation de registres pourra ensuite affecter à un même registre physique plusieurs registres virtuels pour peu que leurs durées de vie ne se recouvrent pas.

Les classes dérivées de `Instr`

Trois classes dérivent de `Instr` :

- `LABEL` (importée en tant que `L` dans la génération de code ARM pour éviter le conflit avec la classe `LABEL` de l'IR) : représente une définition de label. Pour un `LABEL`, `jumps` est vide.
- `MOVE` (`M`) : représente un transfert direct de registre à registre. `jumps` est vide, et cette instruction pourra être éliminée par la suite si la source et la destination correspondent au même registre physique.
- `OPER` (`O`) : une opération classique, ni `LABEL` ni `MOV`.

Le visiteur

Le visiteur est un peu particulier, car on lui fait renvoyer des types différents selon qu'on visite un `Stm` ou un `Sxp` :

- La traduction d'un `Stm` renvoie une liste de `Instr` (des instructions assembleurs encapsulées).
- La traduction d'un `Sxp`, qui signifie qu'on descend dans l'arbre, renvoie une suite d'instruction ainsi qu'un nouveau registre temporaire dans lequel le résultat de l'évaluation de la `Sxp` pourra être trouvé.

Exemple de génération de code (Sxp)

Un accès mémoire peut se coder naïvement ainsi :

```
@visitor(MEM)
def visit(self, mem):
    # Registre pour stocker le résultat
    temp = Temp.create("mem")
    # On évalue l'adresse du MEM
    adr_stms, adr_temp = mem.exp.accept(self)
    # On détermine les instructions à effectuer
    stms = adr_stms + [O("ldr {}, [{}]",
                        dsts=[temp], srcs=[adr_temp])]
    # On retourne les instructions et le résultat
    return stms, temp
```

Exemple de génération de code (Stm)

Une opération MOVE pourra se traduire naïvement ainsi :

```
@visitor(MOVE)
def visit(self, move):
  src_stms, src_temp = move.src.accept(self)
  if isinstance(move.dst, MEM):
    # If we try to move into memory, use a store
    dst_stms, dst_temp = move.dst.exp.accept(self)
    return src_stms + dst_stms + \
      [O("str {1}, [{0}]", srcs=[src_temp, dst_temp])]
  else:
    # The only other possibility is a temporary, this
    # is a move operation
    return src_stms + \
      [M("mov {}, {}",
       dst=move.dst.temp, src=src_temp)]
```

C'est à vous

Il vous faut donc, pour pouvoir passer à l'étape suivante :

- choisir un modèle de jeu d'instruction ARM (par exemple thumb2) ;
- générer le code, avec des registres infinis, pour ce processeur ARM, de la manière la plus efficace possible, en cherchant l'arbre couvrant des instructions IR qui nécessite le moins d'instructions.

Le travail aura lieu dans `arm/gen.py` dans lequel un modèle (modifiable) est donné. Il n'y aura pas de tests automatiques, si ce n'est de vérifier que le code des étapes précédentes fonctionne toujours et que les programmes Tiger peuvent être traduits en assembleur.