Projet de compilation (Étape 4)

Samuel Tardieu
SE202 - Année scolaire 2016/2017

Quatrième 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 step4
```

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

La traduction vers la représentation intermédiaire (IR) est fournie par l'équipe pédagogique. Durant cette étape, vous aurez à découper les instructions en blocs basiques et à les réordonner.

La représentation intermédiaire utilise un nouveau type d'arbre localisé dans `ir/nodes.py`. Certains nœuds vont être détaillés par la suite.

Fonctions intrinsèques

Une fonction intrinsèque est une fonction dont le code est fourni par le système d'exploitation ou la bibliothèque de support de compilateur.

Le programme `tiger.py` et le squelette du *binder* ont été enrichis pour fournir deux fonctions intrinsèques :

- `print_int(n: int)` affiche un entier sur la sortie standard (sans retour chariot) ;
- `exit(n: int)` sort du programme avec un code d'erreur (ou `0` si tout va bien).

De plus, la valeur renvoyée par votre programme principal est implicitement incluse dans une fonction `main`.

Objets supports de l'IR

Deux objets supports sont utilisés dans le cadre de l'IR :

- `Label` représente un nom logique utilisé plus tard dans le cadre de la génération du langage assembleur ;
- `Temp` représente un registre nommé soit à partir d'un registre physique soit à partir d'un registre virtuel avec un suffixe explicatif optionnel.

Ces deux classes disposent d'une méthode `create` permettant de créer un nouvel objet.

Les types de nœuds

Les nœuds de l'arbre intermédiaire sont divisés en deux catégories :

- les nœuds enfants de `Sxp` représentent une expression renvoyant un résultat (*Simple eXPression*, une opération binaire par exemple);
- les nœuds enfants de `Stm` représentent un *StaTeMent*, qui ne renvoie pas de résultat (un transfert de registres par exemple).

Tous les nœuds sont documentés dans `ir/nodes.py`. Leurs noms sont écrits entièrement en majuscules pour les distinguer de deux de l'arbre syntaxique présent dans `ast/nodes.py`.

Les nœuds de l'IR

- `BINOP` (*Sxp*) représente une opération binaire entre deux `Sxp`.
- `CALL` (*Sxp*) représente un appel de fonction (qui renvoie un résultat).
- `JUMP` (*Stm*) représente un saut inconditionnel. Il prend un `Sxp` comme paramètre, car on peut sauter à une adresse calculée par une formule ou une fonction. Si on veut sauter à un label de type `Label`, on l'encapsulera dans un nœud `NAME` qui est de type `Sxp`.

Les nœuds de l'IR

- `CJUMP` (*Stm*) est un saut conditionnel en fonction du résultat d'un opérateur logique binaire. Il prend en paramètres l'opérateur (une chaîne de caractères), les deux opérandes (des `Sxp`) et les deux labels (des `Sxp`) correspondant à l'endroit où sauter si le test est vrai et si le test est faux respectivement.
- `CONST` (*Sxp*) représente une constante entière.
- `SXP` (*Stm*) permet d'indiquer qu'on ne souhaite pas utiliser la valeur de l'expression (`Sxp`) passée en paramètre.

Les nœuds de l'IR

- `LABEL` (*Stm*) est une déclaration de label (qui prend en paramètre un objet de type `Label`).
- `NAME` (*Sxp*) prend également un objet de type `Label` en paramètre et sert à le désigner comme cible d'un saut par exemple : `JUMP(NAME(Label("main")))`.
- `TEMP` (*Sxp*) est un registre (virtuel ou physique) qui prend en paramètre un objet de type `Temp`).
- `MOVE` (*Stm*) est un transfert vers une destination (`Sxp`) d'une source (`Sxp`). Par exemple, pour transférer le résultat de l'opération `3*4` dans le registre physique `r3`, on utilisera : `MOVE(TEMP(Temp("r3")), BINOP("*", CONST(3), CONST(4)))`.

Les nœuds de l'IR

- `SEQ` (*Stm*) prend en paramètres une liste de *statements* (de type `Stm`) et les exécute à la suite.
- `ESEQ` (*Sxp*) prend en paramètres un `Stm` a exécuter en premier, puis un `Sxp` représentant l'expression à évaluer. Par exemple, pour mettre le résultat de `1+2` dans le registre virtual `t_4` et utiliser sa valeur, on utilisera :
  `ESEQ(MOVE(TEMP(Temp("t_4")), BINOP("+", CONST(1), CONST(2))), TEMP(Temp("t_4"))).`

Les nœuds de l'IR

- `MEM` (Sxp) prend en paramètre un `Sxp` et le déréférence. Par exemple, `MEM(fp)` est le contenu de l'adresse pointée par le *frame pointer*, c'est donc le *static link*, soit l'adresse de la *frame* englobante. `MEM(MEM(fp))` est l'adresse de la *frame* englobante deux niveaux au dessus de la fonction courante.

La structure d'une Frame

Un objet de type *Frame* contient :

- les labels (début, fin, endroit où allouer de la place sur la pile, endroit où restaurer les registres) associés à la fonction courante ;
- les entités (registres ou références sur la pile) stockées dans la frame (paramètres, variables locales, registres temporaires) ;
- des utilitaires pour décorer le corps de la fonction avec le code nécessaire pour sauvegarder et restaurer les registres, le pointeur de pile, etc. ;
- des utilitaires pour récupérer l'expression (*Sxp*) correspondant à une entité (registre ou référence sur la pile) définie dans la frame ou prédéfinis, comme le *frame pointer*.

Visualisation de l'IR

Le programme `tiger.py` permet, avec le paramètre `-i`, de visualiser la sortie en langage intermédiaire, en utilisant les registres ARM ou les registres de la machine virtuelle IRVM si `-I` est également donné :

```
% ./tiger.py -iIdE "42"
seq
  label main
  move
    temp rv
    const 42
  label end
seq end
```

Visualisation de l'IR

On remarquera plusieurs choses :

- La machine virtuelle IRVM utilise le registre `rv` pour placer la valeur de retour.
- Les `SEQ` sont affichées entre `seq` et `seq end`.
- Un label `end` indique la fin d'une fonction (équivalent à un `return`). Le nom de ce label (`end` pour `irvm`, dérivé du nom de la fonction pour ARM) se trouve dans le champ `end_label` de la *frame*.

Le code du *dumper* pour l'IR est disponible dans `ir/dumper.py`.

Visualisation de l'IR au format ARM

Si on omet le `-I`, les conventions d'appel sont celles de l'ABI ARM, avec une valeur de retour dans `r0` :

```
% ./tiger.py -idE "42"
seq
  label main
  move
    temp r0
    const 42
  label main$$end
seq end
```

En fait, il y a plus de texte que ça : les mouvements inutiles seront éliminés lors de la génération de code.

Forme canonique

L'arbre n'est pas sous forme canonique prêt à être transformé en assembleur. Par exemple, les appels de fonction (nœuds `CALL`) peuvent prendre d'autres appels de fonction en paramètre. Vous pouvez le visualiser en utilisant par exemple :

```
% ./tiger.py -iIdE "let function f(n: int) = n \
  in f(f(3)) end"
```

Bien entendu, un appel de fonction utilisant les registres, il faudrait plutôt sérialiser les appels. C'est le rôle de la mise sous forme canonique, que l'on peut trouver dans les fichiers `ir/canonical.py` et `ir/hoist.py`.

Forme canonique

On peut effectuer la transformation sous forme canonique en ajoutant `-c` à la ligne de commande (rappel : `--help` ou `-h` permet d'obtenir l'aide) :

```
% ./tiger.py -ciIdE "let function f(n: int) = n \
  in f(f(3)) end"
```

On remarquera que le résultat de `f(3)` est stocké dans un registre intermédiaire avant de procéder au second appel à `f`. De plus, on ne trouve plus de nœuds `ESEQ`, et il n'y a plus qu'un seul nœud `SEQ` par fonction qui représente le code de cette fonction.

Exécution avec IRVM

Le programme `irvm` écrit par Pablo Oliveira permet d'éxécuter le code IR :

```
% ./tiger.py -ciIdE "let function f(n: int) = n*2 \
  in print_int(f(f(3))) end" > /tmp/t.ir
% irvm /tmp/t.ir
12
```

Installation et compilation d'IRVM

```
% cd /tmp
% git clone https://github.com/pablooliveira/irvm
% cd irvm
% ./autogen.sh
% ./configure
% make
% export PATH=$PWD/src:$PATH
```

ou

```
% sudo make install
```

sur votre propre machine (à la place de la dernière commande). Il est possible de devoir installer les paquets `autoconf`, `texinfo`, `flex` et `bison` si vous ne les avez pas en local.

À faire (partie 1)

Il vous faudra vérifier que votre arbre est bien construit en écrivant de petits programmes en Tiger qui devront afficher le bon résultat. Ces programmes devront contenir, sur la première ligne, en commentaire `//`, le résultat attendu, et se trouver dans des fichiers `.tiger` du répertoire `tests` :

- un calcul récursif de factorielle ;
- un calcul non-récursif de factorielle (avec accumulateur) ;
- un calcul de la fonction de Fibonacci (calculez `fibo(15)` par exemple) ;
- des exemples utilisant les boucles ;
- des exemples utilisant les fonctions imbriquées ;
- des exemples utilisant `break`.

Exemple de programme Tiger

Dans le fichier `tests/facta.tiger`, vous pourrez par exemple avoir le calcul de factorielle avec accumulateur :

```
// 3628800
let
  function facta(n: int, a: int) =
    if n < 2 then a else facta(n-1, a*n)
  function fact(n: int) = facta(n, 1)
in
  print_int(fact(10))
end
```

Notation

Une note sera attribuée à la qualité des programmes Tiger écrits et à leur résultat. N'hésitez pas à utiliser également d'autres fonctionnalités du langage. Vous n'êtes pas autorisés à reprendre les programmes de quelqu'un d'autre.

Un petit bonus pourra être accordé à ceux qui trouveront des bugs dans la traduction en langage intermédiaire, et un bonus plus important à ceux qui proposeront des corrections.

Attention à bien comprendre la représentation intermédiaire : c'est au programme du QCM de la semaine prochaine.

À faire (partie 2)

En général, un processeur dispose d'une instruction de saut conditionnel qui permet uniquement de sauter si une condition est vraie. Si elle est fausse, l'exécution continue à l'instruction suivante. On va donc s'arranger pour qu'un saut conditionnel soit toujours suivi immédiatement par sa branche "faux".

Il va vous falloir découper chaque corps de fonction (un `SEQ`) dans la phase de canonisation pour :

- découper l'exécution en blocs basiques ;
- construire des traces ;
- réordonnancer les blocs basiques en fonction des traces.

Blocs basiques

Un bloc basique est un bloc qui :

- commence par un `LABEL` ;
- termine par un `JUMP` ou un `CJUMP` ;
- ne contient pas d'autre `LABEL`, `JUMP` ou `CJUMP`.

Vous allez donc construire un ensemble de blocs basiques indexés par leur label d'entrée.

Découpage en blocs

En parcourant les instructions du `SEQ` qui compose la fonction, vous allez, en commençant par mettre un bloc vide comme bloc courant :

- commencer un nouveau bloc courant quand vous rencontrez un `LABEL` ; si le bloc précédent ne terminait pas par un `JUMP` ou un `CJUMP` (c'est le cas si le bloc courant n'est pas vide), lui ajoutez un `JUMP` vers ce label ;
- terminer le bloc courant s'il termine par un `JUMP` ou un `CJUMP` et commencer un nouveau bloc vide.

Il n'est normalement pas possible que la première instruction d'un bloc ne soit pas un `LABEL` car aucun chemin de code ne pourrait s'y rendre.

Détermination des traces (1/2)

Il faut maintenant déterminer les exécutions possibles pour ordonnancer les blocs au mieux. En commençant par le premier bloc déterminé lors du découpage, et en le plaçant dans une liste de blocs à examiner et en créant une liste des blocs déjà examinés :

- Prendre un bloc dans la liste des blocs à examiner et le placer à la suite des blocs déjà vus, le marquer comme examiné.
- Si le dernier bloc termine par un saut inconditionnel, examiner le bloc cible sauf s'il a déjà été vu, auquel cas on recommence avec un nouveau bloc à examiner.

Détermination des traces (2/2)

- Si le dernier bloc termine par un saut conditionnel, on tente de placer le bloc “faux” à la suite et passer le bloc vrai dans les blocs à examiner. Si ce n’est pas possible, on tente de placer le bloc vrai à la suite en inversant la condition et les labels vrais et faux. Si ce n’est pas possible, on insère un bloc fictif contenant un nouveau label et un sauf inconditionnel vers le label faux, et on remplace le label faux par ce nouveau label dans le saut conditionnel.

On pourra traiter spécialement le bloc commençant par le label de fin de fonction en le marquant comme déjà examiné et en l’insérant explicitement à la fin.

Linéarisation

On peut ensuite concaténer ces blocs qui ont comme propriété que toutes les branches fausses des sauts inconditionnels reviennent à passer à l'instruction suivante.

Lorsqu'un bloc termine par un saut inconditionnel (`JUMP`) au bloc suivant, on supprimera ce saut inutile lors de la concaténation.

Cette concaténation deviendra le nouveau corps de la fonction (nœud `SEQ`).

Travail supplémentaire

Pour ceux qui souhaitent un bonus, un travail supplémentaire est proposé qui touche à toutes les étapes vues jusqu'à présent :

- ajouter un mot clé `return` qui permet de retourner de manière anticipée d'une fonction (avec ou sans valeur) ;
- afficher `return` correctement ;
- lier `return` à la fonction englobante ;
- typer le nœud `return` (qui lui-même est de type *void*) ;
- transformer le nœud `return` en IR (on pourra s'inspirer de la gestion du `break`).

C'est un ajout au langage qui ne nécessite pas d'enrichir l'IR ni les étapes suivantes.

La suite

Dans les séances suivantes, nous aurons à choisir les instructions assembleur correspondant à notre arbre IR pour pouvoir générer du code conforme à l'ABI ARM.

Nous générerons du code pour un nombre infini de registres, puis procèderons à l'allocation de registres pour utiliser les registres physiques ou, lorsque ça sera nécessaire, la pile.