Interprétation et compilation

Chapitre 5

Compilation de langages impératifs

Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/ic

Compilation de langages impératifs

Plan

- Objectif : compilateur pour un langage impératif vers de l'assembleur MIPS.
- Jusque là nous savons :
  - écrire de l'assembleur MIPS,
  - représenter cet assembleur en OCaml,
  - générer du code MIPS depuis une représentation en OCaml,
  - représenter des langages impératifs "haut niveau" en OCaml.
- Pour poursuivre notre chemin vers la compilation, nous devons maintenant voir comment compiler (i.e., traduire) un langage impératif à haut niveau en assembleur.
- Notre objectif va être de traduire notre *représentation intermédiaire* vers votre *représentation de l'assembleur*.

Sémantique

- Pour “traduire” un langage dans un autre, il est nécessaire de donner un sens aux code écrit dans le premier langage, afin de pouvoir en donner une traduction qui aurait le même sens dans le second langage.
- C’est ce qu’on appelle une *sémantique*.

Sémantique

- Pour “traduire” un langage dans un autre, il est nécessaire de donner un sens aux code écrit dans le premier langage, afin de pouvoir en donner une traduction qui aurait le même sens dans le second langage.
- C’est ce qu’on appelle une *sémantique*.
- Dans notre cas on rappelle que ce qui différencie un langage impératif haut niveau de l’assembleur sont :
  - les types de données,
  - les variables,
  - les expressions,
  - les structures de contrôles.

Valeurs de types de bases

- L'essentiel de la sémantique de notre programme aura été analysée lors de la phase d'*analyse sémantique*.
- Notamment, les informations de typage ont majoritairement disparues à cette étape.
- Mais il faut encore compiler les valeurs des types de bases :
  - rien, booléens, entiers, chaînes de caractères, pointeurs, etc.
- La convention en MIPS et de compiler dans le registre `$v0`.

- On représente généralement le rien (ou bien le pointeur nul, la liste vide, etc.) par la valeur 0.
- On utilise donc l'instruction `li` (*load immediate*).
- rien : `li $v0, 0`

- Selon les langages les valeurs booléennes ont leur propre type ou sont définies par convention.
  - En C la valeur 0 représente “faux”, et tout le reste, “vrai”.
  - Dans les langages avec un vrai type booléen, on peut décider de représenter “faux” par 0 en mémoire et “vrai” par 1, pour garder la compatibilité avec le C et se simplifier la tâche.
- faux : `li $v0, 0`
- vrai : `li $v0, 1`

Entiers

- Les entiers se traduisent directement car il s'agit du type "naturel" en assembleur.
- n : `li $v0,` n

Chaînes de caractères

- Pour les chaînes de caractères, la traduction ne peut pas être aussi directe.
- Il est nécessaire de faire une phase de “collecte” des chaînes dans le code pour les placer dans le *segment de données*, en les remplaçant par leur *adresse* en mémoire.
- On utilise donc l’instruction `la` (*load address*).
- “string” : `la $v0, str001`
  avec dans le segment de donnée : `str001: .asciiz "string"`

Variables

- Lors de la compilation, on ne peut pas stocker les valeurs des variables dans un environnement (puisque ces valeurs sont dynamiques).
- Comment faire ?

Variables

- Lors de la compilation, on ne peut pas stocker les valeurs des variables dans un environnement (puisque ces valeurs sont dynamiques).
- Comment faire ?
- Compiler les assignations et stocker les résultats en mémoire.
- Mettre dans notre environnement les *adresses* des valeurs.

- “Stocker en mémoire” signifie écrire sur la *pile*.
- Quels problèmes cela peut poser ?

- “Stocker en mémoire” signifie écrire sur la *pile*.
- Quels problèmes cela peut poser ?
  - Comment avoir des adresses stables pour les variables avec un pointeur de pile qui bouge ?
  - (exemple : accès aux variables hors portée locale)

- "Stocker en mémoire" signifie écrire sur la *pile*.
- Quels problèmes cela peut poser ?
  - Comment avoir des adresses stables pour les variables avec un pointeur de pile qui bouge ?
  - (exemple : accès aux variables hors portée locale)
- Solution : le *frame pointeur*, qui restera stable au sein de chaque fonction.
- On stock donc dans l'environnement pour chaque variable son adresse en mémoire relative à ce pointeur qui sera manipulé dans le registre `$fp`.
- v ← expr : compilation de expr suivi de `sw $v0, ENV[v]`
  où `ENV[v]` est l'adresse de la variable v.
- v : `lw $v0, ENV[v]`
- On détaillera la discipline d'utilisation du *frame pointeur* lorsqu'on parlera de la compilation des fonctions.

Expressions

- Au delà des valeurs et des variables, une expression peut être *composée*.
- On peut représenter ces expressions avec des appels de fonctions, et c'est ce qu'on fera en pratique dans nos petits compilateurs.
- On pourrait aussi avoir certains constructions "natives" dans le langage, qui utilisent directement des instructions assembleurs, mais cela revient à des fonctions systématiquement "inlinées".
- Quoi qu'il en soit, il faut une convention pour passer les arguments aux fonctions.
- On détaillera ces conventions lorsqu'on parlera de la compilation des fonctions.

- Pour bien comprendre la nécessité des conventions, “compilons” l’expression `a+b` :
  - compilation de `a` (dans `$v0`),
    compilation de `b` (dans `$v0`),
    compilation de `+` : `add $v0, A, B`

- Pour bien comprendre la nécessité des conventions, “compilons” l’expression `a+b` :
  - compilation de `a` (dans `$v0`),
    compilation de `b` (dans `$v0`),
    compilation de `+` : `add $v0, A, B`
- Il faut se mettre d’accord sur où stocker les résultats de chaque “sous expressions” au fur et à mesure.
- Le plus simple est d’utiliser la pile. Pourquoi ?

- Pour bien comprendre la nécessité des conventions, “compilons” l’expression `a+b` :
  - compilation de `a` (dans `$v0`),
    compilation de `b` (dans `$v0`),
    compilation de `+` : `add $v0, A, B`
- Il faut se mettre d’accord sur où stocker les résultats de chaque “sous expressions” au fur et à mesure.
- Le plus simple est d’utiliser la pile. Pourquoi ?
  - `(a+b)+(c+d)`

- Une *structure de contrôle* est une instruction qui influe sur le *flot de contrôle* du programme.
- Le flot de contrôle est l'ordre dans lequel les instructions sont exécutées.
  - Il s'agit donc d'une propriété *dynamique*.
  - Le graphe de flot de contrôle d'un programme, représentant tous les flots de contrôles possibles de ce programme, est lui une propriété a priori *statique*.

- Les principales structures de contrôles qu'on retrouve dans les langages impératifs sont :
  - les conditions,
  - les boucles,
  - les fonctions.

Conditions

- À variation de syntaxe prêt, une *condition* a généralement une des formes suivantes :
  - `if (cond) block-then`
  - `if (cond) block-then else block-else`
- Si on note $[\![\mathtt{p}]\!]$ la sémantique du programme `p` et `skip` le programme qui ne fait rien :
  - $[\![\ \texttt{if (cond) block-then}\ ]\!] = [\![\ \texttt{if (cond) block-then else skip}\ ]\!]$
  - $[\![\ \texttt{if (cond) block-then else block-else}\ ]\!] = \begin{cases} [\![\ \texttt{block-else}\ ]\!] \text{ si } [\![\ \texttt{cond}\ ]\!] = 0 \\ [\![\ \texttt{block-then}\ ]\!] \text{ sinon} \end{cases}$

- À variation de syntaxe prêt, une *boucle* a généralement une des formes suivantes :
  - `while (cond) block-do`
  - `for (init ; cond ; next) block-do`
- Toujours avec la même notation pour la sémantique :
  - $[\![\texttt{a ; b}]\!] = [\![\texttt{a}]\!] \circ [\![\texttt{b}]\!]$
  - $[\![\texttt{\{ p \}}]\!] = [\![\texttt{p}]\!]$
  - $[\![\texttt{for (init ; cond ; next) block-do}]\!] = [\![\texttt{init ; while (cond) \{ block-do ; next \}}]\!]$
  - $[\![\texttt{while (cond) block-do}]\!] = \begin{cases} [\![\texttt{skip}]\!] \text{ si } [\![\texttt{cond}]\!] = 0 \\ [\![\texttt{block-do ; while (cond) block-do}]\!] \text{ sinon} \end{cases}$

Fonctions

- Une fonction est *définie* à un endroit et peut être *appelée* à plusieurs.
- Pour donner la sémantique des fonctions on va devoir être plus précis :
  - On va avoir besoin d'un environnement, que l'on mettra en indice de notre fonction : $[\![\_]\!]_E$.
  - E est un ensemble qui contient des associations $nom : valeur$.
  - Si $nom : valeur \in E$ alors $E[nom] = valeur$.
- On va également préciser notre sémantique en lui ajoutant les règles suivantes :
  - $[\![\texttt{v = expr}]\!]_E \circ [\![\texttt{p}]\!]_E = [\![\texttt{p}]\!]_{E \cup \{\texttt{v}:[\![\texttt{expr}]\!]_E\}}$
  - $[\![\texttt{v}]\!]_E = E[\texttt{v}]$

- À variation de syntaxe prêt, une *définition de fonction* est généralement de la forme :
  - `function f (arg`$_1$`, arg`$_2$`, …, arg`$_n$`) block-body`
- Sémantiquement :
  - $[\![$ `function f (arg`$_1$`, arg`$_2$`, …, arg`$_n$`) block-body` $]\!]_E = [\![$ `f = block-body` $]\!]_E$

Appels de fonctions

- À variation de syntaxe prêt, un *appel de fonction* est généralement de la forme :
  - $f(exp_1, exp_2, \ldots, exp_n)$
- Sémantiquement :
  - $[\![f(exp_1, exp_2, \ldots, exp_n)]\!]_E = [\![arg_1 = exp_1 ; arg_2 = exp_2 ; \ldots ; arg_n = exp_n ; f]\!]_E$

En assembleur

- En assembleur les seules structures de contrôles dont on dispose sont des branchements et des sauts.
- Sémantiquement, au niveau de l'assembleur, on ne distingue pas les branchements des sauts.
- Tout ce qui compte c'est que le seul type d'instruction qui influe sur le flot de contrôle permet d'aller directement à une instruction donnée, et que sinon par défaut après toutes les autres instructions on passe à l'instruction suivante dans la mémoire / le code.

Branchements

- Les branchements en assembleur sont de deux types :
  - inconditionnels, qui sautent de toutes façons à l'instruction donnée ;
  - conditionnels, qui sautent seulement quand une condition est remplie.
- En MIPS :
  - inconditionnels : `b`, `j`, `jal`, `jr`, `jalr`, ...
  - conditionnels : `beq`, `beqz`, `bne`, `bgt`, `bge`, ...

- Compiler, c'est traduire un programme en langage haut niveau vers un programme *équivalent* en langage bas niveau.
- Cela suppose entre autre d'être capable d'encoder chacune des structures de contrôle haut niveau en assembleur.
- Si on veut prouver la correction de notre compilateur, il faudrait aussi donner une sémantique à notre langage cible (par exemple l'assembleur), et montrer que la sémantique du programme traduit est la même que celle du programme original.
  - On pourrait par exemple faire en sorte que $[\![\_]\!]_E$ renvoie une fonction qui prend la mémoire dans un certain état et renvoie un nouvel état de la mémoire.
  - Il faudrait alors vérifier que la sémantique de notre programme en assembleur a bien le même comportement (i.e., provoque les mêmes changements en mémoire) que notre programme d'origine.

Conditions

- Pour compiler une condition, il faut :
  - compiler l'expression-test de la condition pour obtenir une valeur booléenne,
  - faire un branchement vers le bon bloc d'instructions.

Conditions

- Dans un langage haut niveau, une condition est une expression arbitraire qui s'évalue vers une valeur booléenne.
- Compiler la condition devrait donc être assez direct.
- L'idée est simplement de mettre le résultat de l'évaluation toujours au même endroit :
  - soit dans un registre spécifique,
  - soit sur le haut de la pile.

Branchements

- Une fois la condition compilée, il faut émettre la bonne instruction de branchement.
- Soit **C[ ]** ma fonction de compilation.
- Si on décide que la compilation des conditions produit toujours leur valeur dans **$v0** :
  - **C[**`if (`*cond*`) `*block-then*` else `*block-else*`]**
    = **C[***cond***]**
    `beqz $v0, else`*XXX*
    C[*block-then***]**
    `b endif`*XXX*
    `else`*XXX*`:`
    C[*block-else***]**
    `endif`*XXX*`:`

Boucles

- Les boucles fonctionnent sur le même principe que les conditions.
  - **C[**`while (`*cond*`) `*block-do*`]**
    = `loop`*XXX*`:`
    C[`cond`**]**
    `beqz $v0, endloop`*XXX*
    C[`block-do`**]**
    `b loop`*XXX*
    `endloop`*XXX*`:`

Fonctions

- Pour compiler une fonction, il faut :
  - compiler le code de la définition de la fonction,
  - compiler les appels de la fonction.

Conventions d'appels

- Quand on écrit une fonction, on ne sait pas où elle va être appelée.
  - Il n'est par exemple pas possible de savoir quels registres seront disponibles au moment de l'exécution de la fonction.
- Par ailleurs, on ne peut pas deviner comment est implémentée une fonction qu'on appelle.
  - Il est pourtant nécessaire de lui passer ses arguments de la manière dont elle les attend.
- Il est donc nécessaire de mettre en place des *conventions d'appels* :
  - pour le passage des arguments,
  - pour la sauvegarde des registres.

Conventions d'appels en MIPS

- En MIPS, des conventions existent déjà.
- Pour les arguments :
  - on passe les arguments dans les registres `$a0` à `$a4`,
  - on passe les arguments suivants (si besoin) sur la *pile*,
  - les valeurs de retours sont dans les registres `$v0` et `$v1`.
- Pour les registres, il y en a deux sortes :
  - ceux sous la responsabilité de l'appelant,
  - ceux sous la responsabilité de l'appelé.

Registres sous la responsabilité de l'appelant

- Les registres `$v0` et `$v1` ainsi que `$a0` à `$a4`, et `$t0` à `$t9` peuvent être modifiés par les fonctions appelées.
- Si une fonction a besoin de conserver leur valeur au travers de l'appel d'une autre fonction :
  - elle doit donc les sauvegarder avant l'appel,
  - et les rétablir après pour pouvoir continuer de les utiliser.

Registres sous la responsabilité de l'appelé

- Les registres `$s0` à `$s7` ainsi que `$gp`, `$sp`, `$fp`, et `$ra` ne doivent pas être modifiés par une fonction appelée.
- Si une fonction a besoin de les utiliser :
  - elle doit donc les sauvegarder avant,
  - ne pas oublier de les rétablir avant de terminer.

- La *pile*, c'est la zone de la mémoire où on va pouvoir faire ces sauvegardes.
- On l'appelle ainsi car on la gère comme une pile :
  - le registre **`$sp`** (stack pointer) est un pointeur sur le haut de la pile,
  - c'est lui qu'on va incrémenter et décrémenter pour savoir où on en est dans la pile,
  - on accédera aux variables sur la pile via ce pointeur.

Empiler

- Attention, la pile grandit vers le bas !
- Pour empiler la valeur de `$t0`, on fait donc :
  - ```
    addi $sp, $sp, -4
    sw $t0, 0($sp)
    ```
- Si on devait empiler `$t0`, `$t1`, `$t2` :
  - ```
    addi $sp, $sp, -12
    sw $t0, 0($sp)
    sw $t1, 4($sp)
    sw $t2, 8($sp)
    ```

Dépiler

- Pour dépiler, c'est pareil dans l'autre sens :
  - ```
    lw $t0, 0($sp)
    addi $sp, $sp, 4
    ```
- Ou encore :
  - ```
    lw $t0, 0($sp)
    lw $t1, 4($sp)
    lw $t2, 8($sp)
    addi $sp, $sp, 12
    ```

Mise en œuvre des conventions

- Pour appeler une fonction, on doit donc :
  - empiler l'état des registres sous la responsabilité de l'appelant (seulement si on en a l'utilité),
  - appeler la fonction (avec `jal`),
  - rétablir l'état de ces mêmes registres en les dépilant.
- À la définition d'une fonction, on doit donc :
  - empiler l'état des registres sous la responsabilité de l'appelé (seulement ceux qu'on va modifier),
  - réserver de l'espace sur la pile pour les variables locales,
  - implémenter le corps de la fonction,
  - libérer l'espace sur la piles des variables locales,
  - rétablir l'état de ces mêmes registres en les dépilant,
  - retourner à la fonction appelée (avec `jr $ra`).

- La *frame* d'une fonction est l'espace réservé sur la pile lors de l'activation (l'appel) de cette fonction.
- Cela comprends :
  - les sauvegardes de registres,
  - l'espace pour les variables locales.
- L'idée est d'utiliser le *frame pointer* (`$fp`) pour sauvegarder l'adresse de cet espace et pouvoir ainsi avoir un repère fixe (contrairement au *stack pointer*) pour celui-ci.
  - (Cela veut notamment dire que le registre `$fp` doit être sauvegardé comme `$ra`.)

Démonstration

- Compilons à la main, puis comme le ferai notre compilateur, le code suivant :

```
1 int factloop (int n)
2 {
3   int r = 1;
4   while (n > 0) {
5     r = n * r;
6     n = n - 1
7   }
8   return r;
9 }
```