Génération de code-I
Constructions élémentaires
28 septembre 2011

Où en est-on ?

- L'analyse lexicale et syntaxique permet de construire un premier arbre de syntaxe abstraite et d'éliminer des programmes incorrects.
- L'analyse sémantique travaille par récurrence sur l'arbre de syntaxe abstraite pour calculer de nouvelles informations (portée, types, surcharge . . .) qui sont conservées dans l'arbre ou ailleurs.
  De nouvelles erreurs sont détectées.
- D'autres représentations comme le graphe de flots collecteront des informations liées à l'exécution du programme.
- La génération de code produit un nouveau code exécutable en machine.

Plan du cours

Génération de code pour de l'assembleur MIPS

- *Introduction*: modèle d'exécution à l'aide d'une pile
- *Code intermédiaire: modèle à pile*
  - Instructions;
  - Cas de base;
  - Compilation des expressions conditionnelles.
- *Appel de procédures*
- *Langages objets*
- *Langages fonctionnels*

Sommaire

Modèle d'exécution du code

- la taille du code est connue à la compilation
- une partie de la mémoire est réservée aux instructions du programme
- un pointeur (*pc*) indique où on en est dans l'exécution
- le code s'exécute de manière séquentielle :
  - sauf instruction explicite de saut, les instructions du programme sont exécutées l'une après l'autre.

Modèle d'exécution, représentation des données

- Les données du programme sont stockées soit dans la mémoire soit dans les registres de la machine.
- La mémoire est organisée en mots (32 ou 64 bits), on y accède par une adresse qui est représentée par un entier.
- Les valeurs simples sont stockées dans une unité de la mémoire.
- Les valeurs complexes sont stockées dans des cases consécutives de la mémoire (structures, tableaux), ou dans des structures chaînées (une liste est représentée par la valeur de son premier élément associée à l'adresse du reste de la liste).
- Les registres permettent d'accéder rapidement à des données simples.
- Le nombre de registres dépend de l'architecture de la machine et est limité.

Allocation

- Certaines données sont explicitement manipulées par le programme source par l'intermédiaire de variables.
- D'autres données sont créées par le compilateur :
  - valeurs intermédiaires lors de l'exécution d'un calcul arithmétique
  - variables lors de l'appel de fonctions
  - valeurs objet, valeurs fonctionnelles ...
- Certaines données ont une durée de vie connue à la compilation: règles de portées du langage, analyse des variables vivantes.

Allocation-suite

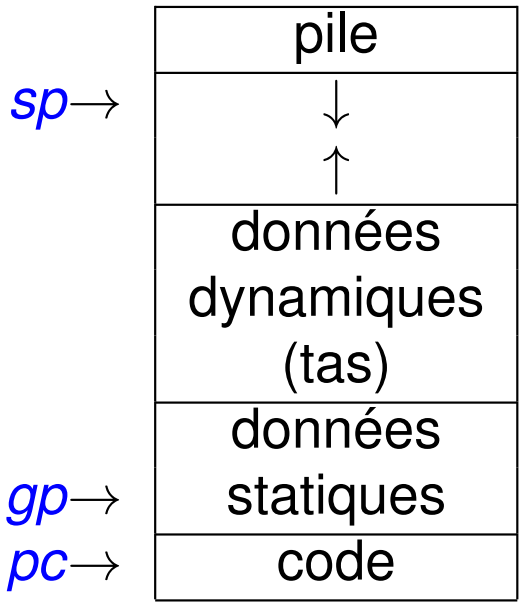
- Les variables globales du programme peuvent être allouées à des adresses fixes, à condition de connaître leur taille.
- La gestion des variables locales des blocs et des procédures, ou bien le stockage de valeurs intermédiaires peut se faire en allouant de la mémoire en pile (stack).
- D'autres données ont par contre une durée de vie qui n'est pas connue à la compilation.
  C'est le cas lorsque l'on manipule des pointeurs (expressions dont la valeur est une adresse).
  Ces données vont être allouées en général dans une autre partie de la mémoire, organisée en tas (heap).

Mémoire

L'espace réservé dans le tas devra être libéré s'il n'est plus utilisé.

- commandes explicites pour libérer l'espace (C, Pascal)
- l'exécution du programme utilise une programme général de récupération de mémoire appelé communément gc pour garbage collector qui est traduit en glaneur de cellules ou ramasse-miettes (Caml, Java).

Schéma d'organisation de la mémoire



Code intermédiaire

Code indépendant de la machine cible.

- factoriser une grande partie du travail de compilation;
- rendre le compilateur plus portable.

Le choix d'un langage intermédiaire est très important.

- assez riche pour permettre un codage aisé des opérations du langage sans créer de trop longues séquences de code
- assez limité pour que l'implantation finale ne soit pas trop coûteuse.

Langage assembleur

- Nombre de registres limité, indispensables pour les calculs.
- Nécessité de sauvegarder certaines valeurs dans une pile.
- Le passage du langage de haut niveau vers de l'assembleur passe par plusieurs langages intermédiaires.
- L'assembleur lui-même utilise des pseudo-instructions de plus haut niveau que le code machine et des étiquettes symboliques.

Sommaire

Notations

Nous spécifierons une fonction *code* qui prend comme argument un arbre de syntaxe abstraite pour le langage et renvoie une suite d'instructions du code de la machine à pile.
Notations concrètes pour représenter la syntaxe abstraite:

- Si $E ::= E_1 + E_2$ est une règle de grammaire nous écrirons
  $code(E_1 + E_2) = \ldots code(E_1) \ldots code(E_2)$
  spécifie la valeur de la fonction *code* sur un arbre de syntaxe abstraite correspondant à l'addition de deux expressions.
- Si C_1 et C_2 représentent des suites d'instructions alors $C_1 \mid C_2$ représente la suite d'instructions obtenue en concaténant C_2 à la suite de C_1.
- La liste vide d'instructions est représentée par [].

Machine à pile

- Stocker les valeurs intermédiaires dans la pile
- Echanger les valeurs entre mémoire et registres pour les calculs
- La pile peut contenir des entiers, des flottants ou des adresses.
- Certaines données grosses en taille comme les chaînes de caractères sont allouées dans un espace supplémentaire, lorsqu'elle devra manipuler une chaîne, la pile se contentera de manipuler l'adresse de la chaîne.

Macros instructions pour la machine à pile

- un registre $sp (stack pointer) pointe sur la première cellule libre de la pile.
- L'adressage se fait par octet : un mot de 32 bits tient sur 4 octets.

Fonction de sauvegarde et de chargement sur la pile:

```
let pushr r = sub $sp, $sp, 4 |
              sw r, 0($sp)

let pop r = lw r, 0($sp) |
            add $sp, $sp, 4
```

Sommaire

Schéma de compilation des expressions

Le résultat de l'expression est stocké dans le registre $a0.
Les valeurs intermédiaires sont stockées dans la pile.

Le code correspondant est donné par :

$$
\begin{aligned}
\mathit{code}(\texttt{entier}) &= \textbf{li}\ \$a0\ \texttt{entier} \\
\mathit{code}(E_1 + E_2) &= \mathit{code}(E_1) \mid \mathit{pushr}\ \$a0 \mid \\
&\quad \mathit{code}(E_2) \mid \ \mathit{popr}\ \$t0 \mid \\
&\quad \textbf{add}\ \$a0\ \$t0\ \$a0
\end{aligned}
$$

- Ce code est très inefficace...
- On peut l'améliorer en utilisant les registres temporaires $\$ti$ ($i = 0 \ldots 9$) et $\$si$ ($i = 0 \ldots 7$)
- Peut se faire naivement (en passant en argument une pile des registres disponibles) ou par des analyses fines qui seront vues plus tard.

Génération de code

La fonction de génération de code est récursive sur l'ast des expressions arithmétiques.

Invariant

Exécution $code(E)$ à partir d'un état où $sp = n$:

- la valeur de l'expression E est calculée dans le registre $a0
- les valeurs de P[m] pour $m < n$ n'ont pas été modifiées

Sommaire

Variables globales/locales

Les instructions **lw** et **sw** permettent de déplacer des valeurs entre une adresse mémoire et un registre.

- **sw**: affecter une valeur calculée dans un registre à un espace réservé pour les variables globales;
- **lw**: charger dans un registre une valeur contenue dans un variable globale.

Notation Si $\$r$ est un registre dont la valeur est une adresse a dans la pile et n est une constante entière alors $n(\$r)$ désigne la valeur à l'adresse $a + n$.

Code pour les variables globales

$$E ::= \texttt{ident}$$

Une variable globale x pourra être stockée dans la zone de données associée à une étiquette symbolique $\texttt{et}(x)$.

```
.data
etiq_x:
    .word 0
.text
```

$$\mathit{code}(\text{id}) \quad = \textbf{lw } \$a0 \ \texttt{et}(\text{id})$$

On peut aussi repérer une variable (globale ou locale) x par son décalage $\texttt{adr}(x)$ par rapport à un pointeur global $\$gp$.

$$\mathit{code}(\text{id}) \quad = \textbf{lw } k(\$gp) \qquad \text{si k=}\texttt{adr}(\text{id})$$

La place nécessaire aux variables est réservée avant l'exécution en retirant à $\$sp$, 4 fois le nombre de variables à stocker.

Affectations

Un langage dont les seules instructions (non-terminal I) sont des suites d'affectations.

Règles de grammaire

$$
\begin{array}{lll}
I & ::= & \epsilon \\
I & ::= & I_1\ A; \\
A & ::= & \texttt{id} \coloneqq E
\end{array}
$$

Génération de code correspondante

$$
\begin{array}{lll}
code(\epsilon) & = [] & \\
code(I_1\ A;) & = code(I_1) \mid code(A) & \\
code(\texttt{id} \coloneqq E) & = code(E) \mid \textbf{sw}\ \$a0\ \texttt{et}(id) & \text{si id global} \\
 & \quad code(E) \mid \textbf{sw}\ \$a0\ k(\$gp) & \text{si k=}\texttt{adr}\text{(id)}
\end{array}
$$

Sommaire

Instructions conditionnelles

- Les expressions conditionnelles et les boucles utilisent des commandes pour effectuer des sauts dans la zone d'instructions.
- Les instructions seront désignées par des adresses symboliques insérées dans le code:
  `label:` introduit un adresse symbolique `label` correspondant au numéro de l'instruction.
- branchement inconditionnel: **j** `label`
- branchement conditionnel si $\$r = 0$: **beqz** $\$r$ `label`

Convention pour la représentation des booléens :

- `false` est représenté par 0
- `true` est représenté par 1 ou par n'importe quel entier positif.

Code conditionnel/boucles

$$
\begin{aligned}
I &::= \textbf{if } E \textbf{ then } I \mid \textbf{if } E \textbf{ then } I_1 \textbf{ else } I_2 \\
I &::= \textbf{while } E \textbf{ do } I \textbf{ done}
\end{aligned}
$$

code(**if** E **then** I)
$= code(E) \mid$ **beqz** $a0 `new-suiv` $\mid code(I) \mid$ `new-suiv:`

code(**if** E **then** I_1 **else** I_2)
$= code(E) \mid$ **beqz** $a0 `new-faux` $\mid code(I_1) \mid$ **j** `new-suiv`
$\mid$ `new-faux:` $code(I_2) \mid$ `new-suiv:`

code(**while** E **do** I **done**)
$=$ `new-loop:` $code(E) \mid$ **beqz** $a0 `new-suiv` $\mid code(I)$
$\mid$ **j** `new-loop` $\mid$ `new-suiv:`

Nouvelles étiquettes `new-faux`,`new-loop`, `new-suiv`.

Compilation d'expressions booléennes

$$
\begin{array}{lll}
B & ::= & B_1 \textbf{ or } B_2 \mid B_1 \textbf{ and } B_2 \mid \textbf{not } B_1 \\
B & ::= & \textbf{true} \mid \textbf{false} \\
B & ::= & E_1 \textbf{ relop } E_2
\end{array}
$$

Compiler les expressions booléennes comme des expressions arithmétiques. Empiler les valeurs 0 ou 1 en utilisant les opérations arithmétiques pour simuler les opérations booléennes.

$$
\begin{array}{ll}
\mathit{code}(\textbf{true}) & = \textbf{li } \$a0\ 1 \\
\mathit{code}(\textbf{false}) & = \textbf{li } \$a0\ 0 \\
\mathit{code}(B_1 \textbf{ and } B_2) & = \mathit{code}(B_1) \mid \texttt{pushr } \$a0 \\
 & \quad \mid \mathit{code}(B_2) \mid \texttt{popr } \$t0 \mid \textbf{mul } \$a0, \$t0, \$a0 \\
\mathit{code}(\textbf{not } B_1) & = \mathit{code}(B_1) \mid \textbf{neg } \$a0 \mid \textbf{add } \$a0, \$a0, 1
\end{array}
$$

Schéma de compilation conditionnel

Utilisation de conditions.

$$
\begin{array}{ll}
B_1 \textbf{ and } B_2 & = \textbf{if } B_1 \textbf{ then } B_2 \textbf{ else faux} \\
B_1 \textbf{ or } B_2 & = \textbf{if } B_1 \textbf{ then vrai else } B_2 \\
\textbf{not } B_1 & = \textbf{if } B_1 \textbf{ then faux else vrai}
\end{array}
$$

Dans un langage avec effets de bord, le fait de calculer ou non les sous-expressions d'une expression booléenne peut avoir des comportements très différents.

Schéma de compilation avec saut

- Les expressions booléennes servent souvent à des opérations de contrôle.
- On peut se donner a priori deux étiquettes `E-vrai` et `E-faux` et compiler l'expression booléenne sans calculer de valeur:
  - le résultat de l'exécution de *E* aboutit à *pc* = `E-vrai` lorsque la valeur de *E* est **vrai** et à `E-faux` sinon.
- On définit donc une fonction *code-bool* qui prend comme argument l'expression booléenne et les deux étiquettes et qui renvoie le code de contrôle.

Code engendré

$$\textit{code-bool}(\textbf{true}, e_v, e_f) = \textbf{j}\ e_v$$
$$\textit{code-bool}(\textbf{false}, e_v, e_f) = \textbf{j}\ e_f$$
$$\textit{code-bool}(\textbf{not}\ B_1, e_v, e_f) = \textit{code-bool}(B_1, e_f, e_v)$$
$$\textit{code-bool}(B_1\ \textbf{or}\ B_2, e_v, e_f) =$$
$$\quad \textit{code-bool}(B_1, e_v, \texttt{new-e}) \mid \texttt{new-e}: \textit{code-bool}(B_2, e_v, e_f)$$
$$\textit{code-bool}(B_1\ \textbf{and}\ B_2, e_v, e_f) =$$
$$\quad \textit{code-bool}(B_1, \texttt{new-e}, e_f) \mid \texttt{new-e}: \textit{code-bool}(B_2, e_v, e_f)$$
$$\textit{code-bool}(E_1\ \textbf{relop}\ E_2, e_v, e_f) =$$
$$\quad \textit{code}(E_1) \mid \texttt{pushr}\ \$a0 \mid \textit{code}(E_2) \mid \texttt{popr}\ \$t0 \mid$$
$$\quad \textit{code}(\textbf{relop})\ \$a0, \$t0, \$a0 \mid \textbf{beqz}\ \$a0\ e_f \mid \textbf{j}\ e_v$$

`new-e` : nouvelle étiquette
code(**relop**) l'instruction de comparaison pour l'opérateur **relop**: **sle**, **slt**, ...

Compilation du test

- On évite l'empilement de valeurs intermédiaires

$$
\begin{array}{l}
\mathit{code}(\textbf{if } E \textbf{ then } I_1 \textbf{ else } I_2) = \\
\quad \mathit{code\text{-}bool}(E, \texttt{new-vrai}, \texttt{new-faux}) \\
\quad \mid \texttt{new-vrai:} \mathit{code}(I_1) \mid \textbf{j } \texttt{new-suiv} \\
\quad \mid \texttt{new-faux:} \mathit{code}(I_2) \mid \texttt{new-suiv:}
\end{array}
$$