Compilation

CM6 - Génération de code d'adressage

ISTIC, Université de Rennes 1
Sebastien.Ferre@irisa.fr

COMP, M1 info

Plan

Plan

Expressions d'adressage

- Une expression d'adressage A désigne une adresse mémoire
  - une expression arithmétique E désigne un entier
  - une expression booléenne B désigne un branchement conditionnel
- Une adresse mémoire est un endroit où :
  - lire : $E \rightarrow A$ — expression atomique
  - écrire : $S \rightarrow A = E$ — affectation
- Expressions atomiques : *id*, désigne une variable
- Expressions complexes :
  - `T[i]` : désigne une cellule de tableau
  - `s.f` : désigne un champ de structure
  - `*p` : désigne une mémoire référencée par un pointeur

Expressions d'adressage

- Une expression d'adressage A désigne une adresse mémoire
  - une expression arithmétique E désigne un entier
  - une expression booléenne B désigne un branchement conditionnel
- Une adresse mémoire est un endroit où :
  - lire : $E \rightarrow A$ — expression atomique
  - écrire : $S \rightarrow A = E$ — affectation
- Expressions atomiques : *id*, désigne une variable
- Expressions complexes :
  - `T[i]` : désigne une cellule de tableau
  - `s.f` : désigne un champ de structure
  - `*p` : désigne une mémoire référencée par un pointeur

Extension de BABIL

Expressions d'adressage :

$A \rightarrow id \mid A[E] \mid A.id \mid *A$

- définition récursive : tableaux, structures et pointeurs peuvent eux-mêmes êtres désignés par des expressions d'adressage
  - ex : `T[i][j]` = `(T[i])[j]`, `s.f.g`, `**p`
  - ex : `(*p)[i]`, `(*p).f`, `*(T[i])`
- nécessitent une vérification de type !
  - ex : `*(T[i])` implique $\frac{T : tableau(entier, pointeur(\tau))}{*(T[i]) : \tau}$

Extension de BABIL

Utilisation des expressions d'adressage :

$$\begin{array}{lcl} E & \rightarrow & A \\ & | & \&A \mid *E \\ S & \rightarrow & A = E \end{array}$$

- les *id* de variables sont remplacées par des expressions d'adressage A
  - comme expression atomique (lecture)
  - comme lieu d'affectation (écriture)
- $\&A$ permet d'utiliser une adresse mémoire comme valeur arithmétique, et $*E$ permet l'inverse
  - manipulation bas niveau et dangereuse (bugs et virus)
  - ex : `p = &x; y = *(p + 4);`
  - suppose conversions entiers/pointeurs !
  - possible en C, rarement dans autres langages

Tableaux

- type $tableau(n, B)$
  - $A = entier$: ici, indices entiers, à partir de 0
  - B : type des éléments
  - n : nombre d'éléments (utile pour allocation et calculs d'adressage)
- schéma mémoire : . . . . . .

- si A est l'adresse du 1er élément du tableau alors $A + i \times taille(B)$ est l'adresse de $A[i]$
- $taille(B)$ est la taille mémoire des valeurs de type B
  - en mots mémoires (ex : 4 octets en 32 bits)
  - $taille(entier) = 1$
  - $taille(tableau(n, B)) = n \times taille(B)$

Tableaux de tableaux

On peut avoir des tableaux de tableaux

- tableaux à plusieurs dimensions
- ex : $A.type = tableau(3, tableau(2, entier))$
- schéma mémoire : .....

- $A[2] = A + 2 \times taille(tableau(2, entier)) = A + 2 \times 2 = A + 4$
- $A[2][1] = A[2] + 1 \times taille(entier) = A[2] + 1 = A + 5$

Remarque

Pour ces calculs d'adresses, on a besoin du type des variables ! connus à la compilation

Génération de code pour les tableaux

. . . . . .

Grammaire attribuée pour les tableaux

$$
\begin{array}{lcl}
A & \rightarrow & A'\ [\ E\ ] \\
 & & \left\{ \begin{array}{lcl}
A.type & := & typeValDeTableau(A'.type) \\
A.place & := & nouvar() \\
A.code & := & A'.code \\
 & @@ & E.code \\
 & @@ & (const\ taille(A.type), A.place, \_, \_) \\
 & @@ & (*, A.place, E.place, A.place) \\
 & @@ & (+, A.place, A'.place, A.place)
\end{array} \right.
\end{array}
$$

Variables

- emplacements dont l'adresse est connue à la compilation
- jouent le rôle de constantes pour les expressions d'adressage A

Génération de code pour les variables : . . . . . .

Grammaires attribuées pour les variables

$$A \rightarrow id \quad \begin{cases} A.type & := & TS.type(id.vallex) \\ A.place & := & nouvar() \\ A.code & := & (\&, A.place, TS.place(id.vallex), \_) \end{cases}$$

Exemple de génération pour une expression d'adressage

A = `tab[i][j+1]`

avec `tab : tableau(3, tableau(2, entier))`,

`i : entier, j : entier.....`

Génération de code pour les lectures/écritures

. . . . . .

Grammaire attribuée pour les lectures/écritures

$$
\begin{array}{lcl}
E & \rightarrow & A \\
 & & \left\{ \begin{array}{lll} E.place & := & nouvar() \\ E.code & := & A.code \\ & @@ & (*_D, E.place, A.place, \_) \end{array} \right. \\
S & \rightarrow & A = E \\
 & & \left\{ \begin{array}{lll} S.code & := & E.code \\ & @@ & A.code \\ & @@ & (*_G, A.place, E.place, \_) \end{array} \right.
\end{array}
$$

Exemple d'instruction avec expressions d'adressage

S = `tab[i] = 2 * tab[i]`

avec `tab : tableau(10,entier), i : entier.....`

Structures

- type $struct(a_1 : \tau_1, \ldots, a_n : \tau_n)$
  - a_i : nom de champ
  - τ_i : type de champ
- schéma mémoire : .....

- adresse des champs par déplacement
  - $A.a_1 = A$
  - $A.a_2 = A + taille(\tau_1)$
  - $A.a_3 = A + \underline{taille(\tau_1) + taille(\tau_2)}$
  - ...

A + déplacement

- déplacement : connu à la compilation (pas A)
  - somme des tailles des types de champs
  - $TS.depl(a_i) = \Sigma_{k=1}^{i-1} taille(\tau_k)$
  - $taille(struct(a_1 : \tau_1, \ldots, a_n : \tau_n)) = \Sigma_{i=1}^{n} taille(\tau_i)$

Génération de code pour les structures

. . . . . .

Grammaire attribuée pour les structures

$$
\begin{array}{lcl}
A & \rightarrow & A' \ . \ id' \\
 & & \left\{ \begin{array}{lcl}
A.type & := & typeChampDeStruct(A'.type, id'.vallex) \\
A.place & := & nouvar() \\
A.code & := & A'.code \\
 & @@ & (const\ TS.depl(A'.type, id'.vallex), A.place, \_, \_ \\
 & @@ & (+, A.place, A'.place, A.place)
\end{array} \right.
\end{array}
$$

Pointeurs

- type *pointeur*(τ)
- schéma mémoire : . . . . . .

- si A est l'adresse d'un pointeur sur un τ
  alors $*A$ est l'adresse d'un τ

Génération de code pour le déréférencement de pointeurs :

. . . . . .

Grammaire attribuée pour les pointeurs

$$A \rightarrow * A'$$

$$\begin{cases} A.type & := & typeValDePointeur(A'.type) \\ A.place & := & nouvar() \\ A.code & := & A'.code \\ & @@ & (*_D, A.place, A'.place, \_) \end{cases}$$

Opérateur "adresse de" &

- $\&A$ et $*E$ sont des expressions (E) de type *entier*
- permettent l'arithmétique de pointeurs
  - ex : `*(&x + 4)`
- ATTENTION : $\&A$ n'est pas une expression d'adressage
  - car pas un emplacement (où écrire) mais seulement une valeur (à lire)
  - `&x = 4` n'a pas de sens

Génération de code pour $\&A$ et $*E$: . . . . . .

Grammaire attribuée pour l'opérateur "adresse de"

Comparer avec la "lecture" :

$$
\begin{array}{lcl}
E & \to & A \\
 & & \left\{ \begin{array}{lcl} E.place & := & nouvar() \\ E.code & := & A.code \mathbin{@@} (*_D, E.place, A.place, \_) \end{array} \right. \\
 & | & \& A \\
 & & \left\{ \begin{array}{lcl} E.place & := & A.place \\ E.code & := & A.code \end{array} \right. \\
 & | & * E' \\
 & & \left\{ \begin{array}{lcl} E.place & := & nouvar() \\ E.code & := & E'.code \mathbin{@@} (*_D, E.place, E'.place, \_) \end{array} \right.
\end{array}
$$

Syntaxe alternative ?

$$E \to * A \mid A \mid *E$$

en pratique, le contenu prime sur le contenant !

Grammaire attribuée pour l'opérateur "adresse de"

Comparer avec la "lecture" :

$$
\begin{array}{lcl}
E & \rightarrow & A \\
 & & \left\{ \begin{array}{lcl} E.place & := & nouvar() \\ E.code & := & A.code \ @@ \ (*_D, E.place, A.place, \_) \end{array} \right. \\
 & | & \& \ A \\
 & & \left\{ \begin{array}{lcl} E.place & := & A.place \\ E.code & := & A.code \end{array} \right. \\
 & | & * \ E' \\
 & & \left\{ \begin{array}{lcl} E.place & := & nouvar() \\ E.code & := & E'.code \ @@ \ (*_D, E.place, E'.place, \_) \end{array} \right.
\end{array}
$$

Syntaxe alternative ?

$$E \rightarrow * \ A \mid A \mid *E$$

en pratique, le contenu prime sur le contenant !

Exemple complet

S = `(*y).next = x; x = y;` avec :

- `typedef struct liste { int val; struct liste *next }`
- `liste *x, *y;` pointeurs sur listes chainées

.....

Optimisation de la lecture des variables

- Le code généré pour $E \Rightarrow A \Rightarrow id$ est particulièrement inefficace :
  1. avant : code vide !
  2. après : $(\&, t_1, id.place, \_)@@(*_D, E.place, t_1, \_)$! !
- soit on optimise le code 3-adresse généré
  - selon le motif `*&x = x` : "le déréférencement de l'adresse de x, c'est x"
- soit on traite le cas des variables ($A \rightarrow id$) à part des autres expressions d'adressage
  - cas sans calcul d'adressage (adresses constantes)
  - spécialisations des règles utilisant A

Grammaire attribuée optimisée pour les variables

$$
\begin{array}{lcl}
E & \rightarrow & id \\
 & & \left\{ \begin{array}{lcl} E.place & := & TS.place(id.vallex) \\ E.code & := & vide \end{array} \right. \\
S & \rightarrow & id = E \\
 & & \left\{ S.code := E.code \ @@ \ (=, TS.place(id.vallex), E.place, \_) \right. \\
 & & \\
A & \rightarrow & * \ id \\
 & & \left\{ \begin{array}{lcl} A.type & := & typeValDePointeur(TS.type(id.vallex)) \\ A.place & := & TS.place(id.vallex) \\ A.code & := & vide \end{array} \right.
\end{array}
$$