Programmation Dirigée par la Syntaxe (PDS)

CM6 - Génération de code d'adressage

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

PDS, M1 info

Plan

1. Génération de code d'adressage
   - Tableaux
   - Variables
   - Lecture/Écriture
   - Structures
   - Pointeurs

Plan

Expressions d'adressage

- Une expression d'adressage *addr* (A) désigne une adresse mémoire
  - une expression arithmétique *expr* (E) désigne un entier
  - une expression booléenne *cond* (C) désigne un branchement conditionnel
- Une adresse mémoire est un endroit où :
  - lire : *addr* (E) — expression atomique
  - écrire : *addr* $=$ *expr* (S) — instruction d'affectation
- Expressions d'adressage atomiques :
  - `x` : désigne une variable
- Expressions d'adressage 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 *addr* (A) désigne une adresse mémoire
  - une expression arithmétique *expr* (E) désigne un entier
  - une expression booléenne *cond* (C) désigne un branchement conditionnel
- Une adresse mémoire est un endroit où :
  - lire : *addr* (E) — expression atomique
  - écrire : *addr* $=$ *expr* (S) — instruction d'affectation
- Expressions d'adressage atomiques :
  - `x` : désigne une variable
- Expressions d'adressage 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 :

$$
\begin{array}{rcl}
\mathit{addr} & ::= & \mathbf{Var}(\mathit{id}) \\
& | & \mathbf{ArrayElem}(\mathit{addr}, \mathit{expr}) \\
& | & \mathbf{Field}(\mathit{addr}, \mathit{id}) \\
& | & \mathbf{Deref}(\mathit{addr})
\end{array}
$$

- 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 $\dfrac{T : \mathit{Table}(\mathit{Integer}, \mathit{Pointer}(\tau)) \qquad i : \mathit{Integer}}{*(T[i]) : \tau}$

Extension de BABIL

Utilisation des expressions d'adressage :

$$\begin{array}{lll} \textit{expr} & ::= & \textbf{Ref}(\textit{addr}) \\ \textit{stat} & ::= & \textbf{Assign}(\textit{addr}, \textit{expr}) \end{array}$$

- les *id* de variables sont remplacées par des expressions d'adressage *addr*
  - $\textbf{Var}(\textit{id}) \rightsquigarrow \textbf{Ref}(\textbf{Var}(\textit{id}))$
    comme expression atomique (lecture, `load`)
  - $\textbf{Assign}(\textit{id}, \textit{expr}) \rightsquigarrow \textbf{Assign}(\textbf{Var}(\textit{id}), \textit{expr})$
    comme lieu d'affectation (écriture, `store`)

Conversions entiers/adresses

- manipulation bas niveau et dangereuse (bugs et virus)
- ex : `p = &x; y = *(p + 4);`
- possible en C, rarement dans autres langages
- suppose conversions entiers/adresses !

  addr ::= **At**(*expr*, *type*) //`float*(p+8)`

  expr ::= **Address**(*addr*) //`&x`

Tableaux

- type $Tableau(n, \tau)$
  - ici, indices entiers, à partir de 0
  - τ : type des éléments
  - n : nombre d'éléments (utile pour allocation et calculs d'adressage)
- schéma mémoire : (1). . . . . .

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

Tableaux de tableaux

On peut avoir des tableaux de tableaux

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

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

Remarque

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

Génération de code pour les tableaux

(3)……

ASD attribuée pour les tableaux

$$
\begin{array}{lll}
addr & ::= & \textbf{ArrayElem}(addr_1, expr) \\
 & & \left\{ \begin{array}{lll}
type & := & typeValDeTableau(addr_1.type) \\
place & := & nouvar() \\
code & := & addr_1.code \mathbin{@@} expr.code \\
 & @@ & (\texttt{CONST}\ taille(addr.type), addr.place, \_, \_) \\
 & @@ & (*, addr.place, expr.place, addr.place) \\
 & @@ & (+, addr.place, addr_1.place, addr.place)
\end{array} \right.
\end{array}
$$

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 : (4). . . . . .

ASD attribuée pour les variables

$$
\begin{array}{lll}
addr & ::= & \textbf{Var}(id) \\
 & & \left\{ \begin{array}{lll}
type & := & TS.type(id.vallex) \\
place & := & nouvar() \\
code & := & (\&, addr.place, TS.place(id.vallex), \_)
\end{array} \right.
\end{array}
$$

Exemple de génération pour une expression d'adressage

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

avec `tab : Tableau(3, Tableau(2, Integer))`,

`i : Integer, j : Integer`

(5).....

Génération de code pour les lectures/écritures

(6)……

ASD attribuée pour les lectures/écritures

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \textbf{Ref}(\mathit{addr}) \\
 & & \left\{ \begin{array}{lcl} \mathit{expr}.\mathit{place} & := & \mathit{nouvar}() \\ \mathit{expr}.\mathit{code} & := & \mathit{addr}.\mathit{code} \\ & @@ & (\star\mathtt{D}, \mathit{expr}.\mathit{place}, \mathit{addr}.\mathit{place}, \_) \end{array} \right. \\
\mathit{stat} & ::= & \textbf{Assign}(\mathit{addr}, \mathit{expr}) \\
 & & \left\{ \begin{array}{lcl} \mathit{stat}.\mathit{code} & := & \mathit{expr}.\mathit{code} \\ & @@ & \mathit{addr}.\mathit{code} \\ & @@ & (\star\mathtt{G}, \mathit{addr}.\mathit{place}, \mathit{expr}.\mathit{place}, \_) \end{array} \right.
\end{array}
$$

Exemple d'instruction avec expressions d'adressage

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

avec `tab : Tableau(10,Integer), i : Integer` (7).....

Structures

- type $Struct(a_1 : \tau_1, \ldots, a_n : \tau_n)$
  - a_i : nom de champ
  - τ_i : type de champ
- schéma mémoire : (8). . . . . .

- 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

(9)……

ASD attribuée pour les structures

$$
\begin{array}{lll}
addr & ::= & \textbf{Field}(addr_1, id) \\
 & & \left\{ \begin{array}{lll}
addr.type & := & typeChampDeStruct(addr_1.type, id.vallex) \\
addr.place & := & nouvar() \\
depl & := & TS.depl(addr_1.type, id.vallex) \\
addr.code & := & addr_1.code \\
 & @@ & (\texttt{CONST}\ depl, addr.place, \_, \_) \\
 & @@ & (+, addr.place, addr_1.place, addr.place)
\end{array} \right.
\end{array}
$$

Pointeurs

- type *Pointer*(τ)
- schéma mémoire : (10)……

- 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 : (11)……

ASD attribuée pour les pointeurs

$$
\begin{array}{lll}
addr & ::= & \textbf{Deref}(addr_1) \\
 & & \left\{ \begin{array}{lll}
addr.type & := & typeValDePointeur(addr_1.type) \\
addr.place & := & nouvar() \\
addr.code & := & addr_1.code \\
 & @@ & (\texttt{*D}, addr.place, addr_1.place, \_)
\end{array} \right.
\end{array}
$$

Conversions entiers/adresses

- **Address**(*addr*) : *expr* et **At**(*expr*, *type*) : *addr* expriment des conversions entre entiers et adresses
- elles permettent l'arithmétique de pointeurs
  - ex : `*(&x + 8)`
- ATTENTION : **Address**(*addr*) 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 les conversions entiers/adresses

(12)……

ASD attribuée pour les conversions entiers/adresses

Comparer avec la "lecture" :

$$
\begin{array}{lcl}
\textit{expr} & ::= & \textbf{Ref}(\textit{addr}) \\
 & & \left\{ \begin{array}{lcl} \textit{expr.place} & := & \textit{nouvar}() \\ \textit{expr.code} & := & \textit{addr.code} \\ & @@ & (\star\mathtt{D}, \textit{expr.place}, \textit{addr.place}, \_) \end{array} \right. \\
 & | & \textbf{Address}(\textit{addr}) \\
 & & \left\{ \begin{array}{lcl} \textit{expr.place} & := & \textit{addr.place} \\ \textit{expr.code} & := & \textit{addr.code} \end{array} \right. \\
\textit{addr} & ::= & \textbf{At}(\textit{expr}, \textit{type}) \\
 & & \left\{ \begin{array}{lcl} \textit{addr.place} & := & \textit{expr.place} \\ \textit{addr.type} & := & \textit{type} \\ \textit{addr.code} & := & \textit{expr.code} \end{array} \right.
\end{array}
$$

Noter que les conversions ne coutent rien !

Exemple complet

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

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

(13).....

Optimisation de la lecture des variables

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

ASD attribuée optimisée pour les variables

$$
\begin{array}{lcl}
expr & ::= & \mathbf{Ref}(\mathbf{Var}(id)) \\
 & & \left\{ \begin{array}{lcl} expr.place & := & TS.place(id.vallex) \\ expr.code & := & vide \end{array} \right. \\
 & | & \mathbf{Address}(\mathbf{Var}(id)) \\
 & & \left\{ \begin{array}{lcl} expr.place & := & nouvar() \\ expr.code & := & (\&, expr.place, TS.place(id.vallex), \_) \end{array} \right.
\end{array}
$$

$$
\begin{array}{lcl}
stat & ::= & \mathbf{Assign}(\mathbf{Var}(id), expr) \\
 & & \left\{ \begin{array}{lcl} stat.code & := & expr.code \\ & @@ & (=, TS.place(id.vallex), expr.place, \_) \end{array} \right.
\end{array}
$$

$$
\begin{array}{lcl}
addr & ::= & \mathbf{Deref}(\mathbf{Var}(id)) \\
 & & \left\{ \begin{array}{lcl} addr.type & := & typeValDePointeur(TS.type(id.vallex)) \\ addr.place & := & TS.place(id.vallex) \\ addr.code & := & vide \end{array} \right.
\end{array}
$$