TD - Gestion des variables et fonctions

Bogdan Pasca

19 October 2010

Niveaux d'imbrication - variables

Question (i)

Faites tourner sur papier le programme suivant :

```
1  static int s;
2
3  int main(int argc, char** argv) {
4    int t;
5
6    s = 0;
7    for (t=0;t<2;t++) {
8      int t;
9      for (t=0;t<3;t++) {
10       int i;
11       i = s;
12       i++;
13       s = i;
14     }
15     for (t=0;t<3;t++) {
16       int s;
17       int i;
18       i = s;
19       i++;
20       s = i;
21     }
22     s++;
23     t++;
24   }
25   printf("s = %d\nt = %d\n",s,t);
26   return 0;
27 }
```

Question (ii)

Combien de niveaux d'imbrication y a-t-il dans le programme donné ? Annotez chaque définition et utilisation d'une variable (et d'une fonction) avec le niveau correspondant.

Question (iii)

Utilisez l'information trouvée par annotation pour réécrire le programme avec des noms de variables `t_`*niveau_compteur*.

La table des symboles

Un compilateur doit faire le même travail que celui que vous venez de faire. Il utilise pour cela une structure à base de tables de hachage et de pointeurs, que l'on appelle table de symboles.

Une table de hachage est une structure de données avec un coût amorti en $\mathcal{O}(1)$ définie par les deux opérations suivantes :

- *recherche(nom)* qui retourne l'information *info* indexée par *nom* (ou bien $\perp$ si l'index *nom* n'existe pas)
- *insérer(nom,info)* qui insère l'information *info* à l'index *nom*.

Au début de la compilation, la table de symbole ne consiste qu'en une table de hachage vide. A chaque fois que le compilateur rencontre un nouveau niveau d'imbrication, il crée une nouvelle table de hachage qu'il relie à la précédente par un pointeur. Il insère chaque définition d'une variable dans la table de hachage actuelle. Lors qu'il rencontre une référence à une variable, il la recherche dans la table de symbole en remontant récursivement les tables de hachages et en invoquant l'opération *recherche* jusqu'à ce qu'il la trouve dans une de ces tables. La référence à la variable, contenue bien sûr dans l'arbre de syntaxe abstraite, est alors annotée par le niveau de sa définition et son numéro dans la table de hachage correspondante. Quand le compilateur quitte une portée, il remonte au niveau supérieur sans désallouer la table de hachage qu'il vient de créer et de remplir.

Question (iv)

Calculez la table de symbole pour le code donné et annotez les variables comme le ferait le compilateur sur l'arbre de syntaxe abstraite. Comparez avec les noms de variables uniques `t_`x`_`y de la question précédente.

Enregistrements d'activation

Comme on vient de voir, il est possible d'analyser le code source et de donner un nom unique à chaque variable qui y est utilisée. Pourtant, on n'a pas encore donné de méthode qui permette d'associer à chacun de ces noms uniques une case mémoire lors de l'exécution.

Dans un premier temps, nous supposons que chacune de variables du code source est stockée en mémoire d'où elle est lue et où on la réécrit après modification.

Question (v)

Pourquoi n'alloue-t-on pas un tableau de taille fixe pour chaque case remplie d'une des tables de hachage de la table de symboles afin de pouvoir remplacer les noms uniques par des adresses ? Donnez un bout de code dont la sémantique ne serait pas respectée par cette méthode.

Question (vi)

Quelles sont les informations qui doivent être gérées sur la pile d'exécution ?

Question (vii)

L'ensemble des informations nécessaires pour un appel de fonction stockées sur la pile d'exécution est appelé *enregistrement d'activation* (*activation record* ou bien *frame*). Sa structure est souvent fixe pour un type de machine donné. Motivez ce choix. Proposez une structure pour une plateforme imaginaire mais typique qui permette de gérer des appels récursif ainsi que des variables locales et leur accès. On ne considerera pas encore l'accès aux variables des niveaux supérieurs.

Question (viii)

Donnez du pseudo-code assembleur pour traduire l'accès à la variable `i` dans la procédure suivante :

```
1 void f(int x) {
2   int i;
3   i = x + 1;
4 }
```

D'où chargez-vous l'adresse la case mémoire sur la pile pour la variable ? On introduira un registre particulier que l'on appellera *pointeur d'enregistrement d'activation* (*activation record pointer* ARP ou bien *framepointer* FP).

Question (ix)

A quels moments mettra-t-on le framepointer à jour ? Est-ce une tâche effectuée de préférence par l'appelant ou par l'appelé ? Comment le retrouvera-t-on au bout d'une fonction ? A part pour sa restauration, aura-t-on besoin du framepointer de l'appelant encore pour une autre raison ?

Question (x)

Considérez le code Pascal suivant :

```
1  program test;
2  var
3     x, y :  integer;
4
5  procedure printMultCompliquee(x, y : integer);
6  var
7     a, b :  integer;
8
9     function multRec(y :  integer) : integer;
10    var
11       r : integer;
12    begin
13       if y = 0 then
14          r := 0
15       else
16          r := a + multRec(y - 1);
17       multRec := r;
18    end;
19
20 begin
21
```

```
22      a := x;
23      b := multRec(y);
24
25      writeln('x * y = ',b);
26  end;
27
28  begin
29      x := 2;
30      y := 3;
31      printMultCompliquee(x,y);
32  end.
```

Comment pourra-t-on accéder à la variable `a` dans la fonction `multRec` vu que celle-ci est récursive et peut ainsi faire croître la pile sans limite ? On introduira un nouveau pointeur que l'on appellera *lien d'accès* ou bien *lien statique* (*access link*, *static link* ou bien encore *lexical pointer*). Où ce pointeur pointera-t-il alors ?

Question (xi)

Déroulez le programme Pascal ci-dessus et dessinez l'état de la pile ainsi que des pointeurs pour le moment où elle atteint sa taille maximale.

Edition de lien

L'édition de lien est la repartition des tâches pour l'allocation d'un enregistrement d'activation. Elle se décompose en quatre phases :

- **Pre-appel :** Mise en place de l'enregistrement d'activation par l'appelant. Evaluation des paramètres effectifs, allocation de mémoire pour le résultat, empilement du framepointer actuel, du lien statique ainsi que, finalement, de l'adresse de retour.
- **Prologue :** Création de l'espace pour les variables locales par l'appelé. Accès eventuel au variables de niveaux supérieurs en remontant la chaîne des liens statiques.
- **Epilogue :** Libération de la mémoire allouée pour les variables locales, restauration du framepointer, saut de retour à l'adresse de retour se trouvant sur la pile.
- **Post-appel :** Récupération du résultat de la fonction appelée par appelant, libération de la mémoire pour les paramètres.

Question (xii)

Traduisez à la main le code C ci-dessous en un assembleur que vous inventerez en respectant les conventions établies et les quatres phases de l'édition de lien. Pensez aussi à utiliser une méthode standard pour l'évaluation des paramètres.

```
33  int inc(int i) {
34    int s;
35    s = i + 1;
36    return s;
37  }
38  ...
39  c = inc(3 + 4);
```

Question (xiii)

Pourquoi la plupart des assembleurs (même RISC) implémentent-ils une opération `call` qui stocke le compteur ordinal courant sur la pile et fait ensuite un saut vers une adresse donnée ? Il semblerait que les opérations `push` et `jump` traditionelles puissent faire l'affaire aussi. En fait, c'est une question d'architécture.

Niveaux d'imbrication - fonctions

Plusieurs langages (Pascal, Caml) permettent l'imbrication de fonctions. C'est-à-dire, il est possible de définir une fonction à l'intérieur d'une autre fonction, comme on l'a déjà vu avec l'exemple Pascal ci-dessus.

Question (xiv)

Argumentez dans quelle mésure la gestion des niveaux d'imbrication est pareille pour les variables et pour les fonctions. Donnez des exemples simples où on a besoin de téchniques supplémentaires pour linéariser la structure des fonctions, c'est-à-dire sortir les fonctions imbriquées. D'abord, on ne considerera que les langages impératifs de type Pascal sans pointeurs.

Question (xv)

Comparez les deux programmes suivants donnés en syntaxe Caml et Pascal. Considerez aussi leur sortie. Que se passe-t-il ? Expliquez !

```
1  program test;
2
3  type fun = function(x : integer) : integer;
4
5  var
6     g, h : fun;
7     r :  integer;
8
9     procedure instantiate(var f : fun; y : integer);
10    var
11       a :  integer;
12
13       function add(x :  integer) : integer;
14       begin
15          add := x + a;
16       end;
17
18    begin
19       a := y;
20       f := add;
21    end;
22
23 begin
24    instantiate(g,3);
25    instantiate(h,4);
26    r := g(2);
27    writeln(r);
28 end.
```

Sortie :

```
clauter@schorle:~/TDCompil/td6$ ./a.out
6
```

```
let instantiate y =
  let a = y
  in
  let add x = x + a
  in
  add
;;


let g = instantiate 3
in
let h = instantiate 4
in
let r = g 2
in
print_int(r)
;;
```

Sortie :

```
clauter@schorle:~/TDCompil/td6$ ./a.out
5
```

Question (xvi)

Y a-t-il un moyen simple pour remédier au problème que l'on observe pour le programme en Pascal ? Que doit-on faire alors en Caml pour que ça fonctionne ? Quelles sont les conséquences que cela entraîne ?