Gestion mémoire et Représentation intermédiaire

Pablo de Oliveira <pablo.oliveira@uvsq.fr>

March 23, 2015

I

Gestion Memoire

Variables locales

Les variables locales sont stockées:

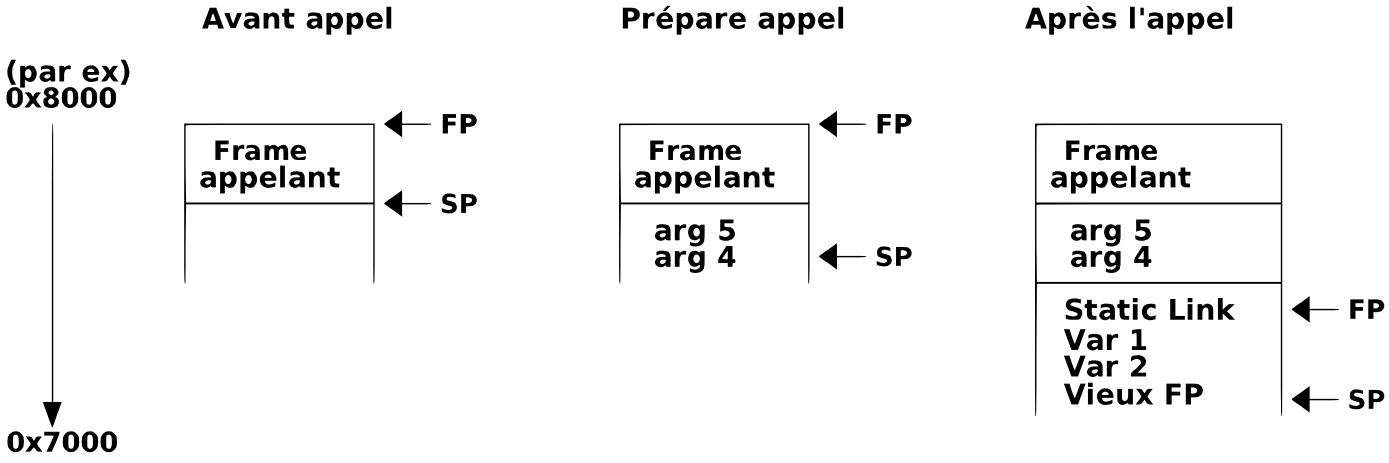
- Soit dans un registre, 1 cycle latence
- Soit sur la pile (mémoire), 2-100 cycles latence

Stratégie: utiliser les registres le plus possible

Problème: pas assez de registres

- L'allocateur de registres maximise le nombre de variables stockées dans des registres
- Si les registres viennent à manquer les variables sont temporairement stockées sur la pile
- Certaines variables *doivent* être stockées sur la pile

Organisation de la pile (Tiger)



Variables qui échappent

- On dit qu'une variable échappe lorsqu'elle peut être accédée en dehors de sa portée.

```
function f1() : int =
    let
        /* a est locale dans f1() ...   */
        var a := 42
        /* ... mais accédée depuis f2() */
        function f2() = a := a + 1
    in
        f2() ; a
    end
```

- L'allocation de registres est locale à une fonction. Après un appel, éventuellement récursif, les registres peuvent changer et ne plus contenir la valeur de a.
- a *échappe* et doit être stockée sur la pile

Détection des variables qui échappent

- Un visiteur d'arbre compare pour chaque variable la profondeur de sa déclaration et de son utilisation
- Une variable déclarée à une profondeur différente échappe

Gestion mémoire des variables échappées

- Le compilateur rajoute un paramètre supplémentaire spécial *static link* à chaque fonction.
- Le *static link* pointe vers le frame pointer de la fonction contenante.
- Il permet au contenu d'accéder aux variables dans la pile du conteneur.

Comment remonter plusieurs scopes ?

```
function f1() : int =
    let
        var a := 42
        function f2() =
            let function f3() = a := a + 1
            in f3() end
    in
        f2() ; a
    end
```

- Ici on doit remonteur deux niveaux d'imbrication.
- Chaque fonction stocke le *static link* de la fonction contenante sur sa pile.
- La fonction `f3()` peut donc remonter la chaîne de *static link* pour accéder à `a` dans `f1()`.

Attention: le static link est *Statiquement* calculé

```
1   function f1() : int =
2       let
3           var a := 42
4           function f2( b : int ) : int =
5               if b = 0 then a else f2(b-1) + a
6       in
7           f2(10)
8       end
```

- L'appel `f2(10)` en ligne 7 est remplacé par `f2(fp, 10)`
- L'appel `f2(b-1)` en ligne 5 est remplacé par `f2(MEM(fp), 10)` et non pas `f2(fp, 10)`. Pourquoi ?

Calcul du Static link

- d = profondeur(appel) - profondeur (appellé)
- si $d < 0$, alors on passe notre SL.
- Si $d >= 0$, alors on remonte $d + 1$ SL

```
f1() {
    f2() {
        f3() {
            f1() ---> niveau 3 appele niveau 1, d = 2
        }             3 niveaux à remonter
    }                 sl = MEM(MEM(MEM(fp)))

    f1()          ---> niveau 1 appele niveau 1, d = 0
                       1 niveau à remonter
                       sl = MEM(fp)

    f2()          ---> niveau 1 appelle niveau 2, d < 0
                       pas de niveau à remonter
}                      sl = fp
```

Exercice

- Quels static links sont passés pour chaque appel de fonction ?

```
1    function f1() : int =
2    let
3        var a := 41
4        function f2() : int =
5            let
6                function f3() : int =
7                    let function f4() =
8                        a := 0
9                    in f4();
10                       f2() end
11            in
12               f2();
13               a
14           end
15   in
16       f2()
```

À quelle adresse mémoire aller chercher a ?

```
1    function f1() : int =
2    let
3        var a := 41
4        function f2() : int =
5            let
6                function f3() : int =
7                    let function f4() =
8                        a := 0
9                    in f4(fp);                          # 3 - 4 < 0
10                   f2(MEM(MEM(fp))) end # 3 - 2 = 1
11           in
12               f2(MEM(fp));                            # 2 - 2 = 0
13               a
14           end
15   in
16       f2(fp)                                          # 1 - 2 < 0
17   end
```

A quelle adresse mémoire aller chercher a ?

```
1    function f1() : int =
2    let
3        var MEM[FP-4] := 41                <---------.   <--.
4        function f2() : int =                         |      |
5            let                                       |      |
6               function f3() : int =                  |      |
7                   let function f4() =                |      |
8        MEM[(MEM(MEM(MEM(fp)))-4] := 0 #---------'    |      |
9                   in f4(fp);                         |      |
10                     f2(MEM(MEM(fp))) end            |      |
11            in                                       |      |
12               f2(MEM(fp));                          |      |
13               MEM[(MEM(fp)-4)] + 1 #----------------'      |
14           end                                              |
15   in                                                       |
16       f2(fp)                                               |
17   end                                                      |
```

Tableaux et structures

- Dans Tiger, les tableaux et structures sont allouées sur le *tas*.
- On utilise un équivalent système de la fonction `malloc()` pour obtenir un nouveau segment de mémoire.
- Une variable tableau ou structure est un pointeur vers le segment alloué.
- Les tableaux et structures sont passés par référence (et non par valeur).
- Dans Tiger, il n'y a pas de désallocation mémoire pour les structures et tableaux.

Implémentation dans Tiger

- La phase de traduction en RI doit pouvoir allouer des variables locales dans une fonction.
- Il est trop tôt pour décider exactement où elles sont stockées:
  - L'allocateur de registres est en fin du compilateur.
  - Mais on ne sait pas toujours si elles seront dans un registre ou sur la pile.
  - L'organisation de la pile est propre à chaque architecture.
- On utilise la classe Abstraite `Frame`
  - On crée un objet `Frame` pour chaque nouvelle fonction
  - On alloue des variables en utilisant `Access Frame.alloc(bool is_escape_var);` qui nous retourne un objet de type `Access`.
  - `Access` nous permet de manipuler des variables locales en déférant à plus tard le choix du stockage (registre, pile).

Comment est implémenté Access

- Si la variable échappe, on sait qu'elle ira sur la pile.
  - L'objet `Frame` réserve une case dans la pile pour stocker la variable
  - `Access.getExp(Exp fp)` retourne une expression de type `Mem(Binop(-, fp, Const(position))`
- Si la variable n'échappe pas, on ne sait pas encore.
  - L'objet `Frame` crée un nom unique, par exemple, `t127`.
  - `Access.getExp(Exp fp)` retourne une expression de type `TempExp(t127)`
  - L'allocateur de registre choisira de stocker `t127` dans un vrai registre ou sur la pile.

Généricité

- Un code bien structuré et modulaire limite les dépendances inutiles.
- La traduction IR n'a pas à ce soucier de l'implémentation de la pile qui est propre à chaque architecture (rôle du Backend).
- Pourtant elle doit créer des Objets `Frame` différents pour chaque Architecture ciblée: `FrameMips`, `FrameArm`.
- Comment faire ?

Design pattern: Factory

- On déclare une interface `Builder` qui permet de fabriquer des objets de type `Frame`.
- Le module de traduction dépend que des interfaces génériques `Builder` et `Frame`
- Mais il reçoit en argument une implémentation concrète de `Builder`, par exemple `ArmBuilder`.
- Maintenant il peut appeler la fonction `Builder.newFrame()` et il obtiendra des Frame de type Arm.
- Cette encapsulation est bonne et facilite la maintenance et le déboggage du projet.
- Parce que le module de traduction ne sait pas le type de Frame qu'il manipule, il ne peut pas aller "fouiller" dans l'implémentation du backend, ce qui produirait du code non-générique, inmaintenable.

II

Traduction en Représentation Intermédiaire

Choix d'une représentation intermédiaire

- Doit être simple et se débarrasser du "sucre syntaxique".
- Doit conserver suffisamment d'informations pour être optimisable.
- Il faut arriver à des compromis.
- eg. Doit-on conserver des expressions de tableau ?
  - Oui: facilite l'analyse de dépendances
  - Non: complique la propagation de constantes, la réduction de force, etc.

Tree

- Langage intermédiaire sous forme d'arbre
- Utilisation de temporaires (infinité de registres)
- Jump conditionnel à deux branches

Grammaire de Tree

```
Exp ::= "const" int
      | name
      | "temp" Temp
      | "binop" Oper Exp Exp
      | "mem" Exp
      | "call" name [{Exp}]
      | "eseq" Stm Exp

Stm ::= "move" Exp Exp
      | "sxp" Exp
      | "jump" name
      | "cjump" Relop Exp Exp name name
      | "seq" [{Stm}]
      | "label" Label

name ::= "name" Label
Oper ::= "+" | "-" | "*" | "/"
Relop ::= "=" | "<>" | "<=" | ">=" | "<" | ">"
```

Exemple de Tree: 1 + 2 * 5

```
seq
  label main      # fonction main
  move            # sauve le FP
    temp x1
    temp fp
  move            # met à jour le FP
    temp fp
    temp sp
  move            # réserve une case sur la pile
    temp sp
    binop (-)
      temp sp
      const 4
  move            # écrit le SL
    mem
      temp fp
    temp i0
  move
    temp rv
    eseq
      sxp         # 1 + 2 * 5
        binop (+)
          const 1
          binop (*)
            const 2
            const 5
      const 0     # retour = 0
  move
    temp sp       # restaure SP
    temp fp
  move
    temp fp       # restaure FP
    temp x1
  label end
seq end
```

Traduction des expressions

- Visiteur AST
  - Parcourir l'arbre en remplaçant les expressions les plus profondes par des arbres traduits.
  - AST -> RI TREE

Traduction des expressions (1/3)

- La traduction dépend du contexte. Exemple: a $<$ b ?
- Si le contexte est `if a < b then ... else ...`

```
cjump(<, a, b, truelabel, falselabel)
```

Traduction des expressions (2/3)

- La traduction dépend du contexte. Exemple: a $<$ b ?
- Si le contexte est `t = a < b`

```
eseq (seq (
        cjump (a < b, ltrue, lfalse),
        label ltrue
              move temp t, const 1
              jump lend
        label lfalse
              move temp t, const 0
        label lend),
     temp t)
```

Traduction des expressions (3/3)

- La traduction dépend du contexte. Exemple: a $<$ b ?
- Si le contexte est `(a < b, ())`

```
seq(sxp(a), sxp(b))
```

Comment résoudre le problème ?

- Notre Visiteur d'AST traduit l'arbre en commençant par les expressions les plus profondes.
- Le contexte n'est pas encore connu lorsque l'on traduit a<b.
- Idée: Retarder la traduction.

Coquilles: Shell

Au lieu de faire la traduction directement on retourne des coquilles, où `Shell`:

- Ex Expression Shell, retarde une expression
- Nx Statement Shell, retarde un statement
- Cx Condition Shell, retarde un test conditionnel

	unNx	unEx	unCx(t,f)
Ex(e)	sxp(e)	e	cjump(<>, e, 0, t, f)
Cx(a<b)	seq(sxp(a), sxp(b)	eseq(t ← (a<b), t)	cjump(<, a, b, t, f)
Nx(s)	s	error	error

Traduction While

```
while condition do
  body
```

```
label test
   cjump(!condition,
         done,
         continue)
label continue
   body
   jump test
label done
```

Exercice: Traduction For

- Comment traduire le programme

```
for i:= low to high do body
```

Exercice: Traduction For

- Comment traduire le programme

```
for i:= low to high do body
```

- Votre traduction marche-t-elle lorsque $high = 2^{31} - 1$?

Exercice: Traduction For

```
let i := min
    limit := max
in
    if (i > limit) goto end
    loop:
       body
       if (i >= limit) goto end
       i = i + 1
    goto loop
end:
```

Optimisations des IFs (1/3)

```
if a | b then tlab else flab
```

est remplacé par

```
if (if a then 1 else b) then tlab else flab
```

qui va être compilé vers une cascade de sauts.

```
        CJUMP(a, lab1, lab2)
    lab1:
        t <- 1
        JUMP(out)
    lab2:
        t <- b
    out:
        CJUMP(t, tlab, flab)
    ...
```

Optimisations des IFs (2/3)

Ce serait plus efficace de remplacer

```
if (if a then 1 else b) then tlab else flab
```

par

```
if a then tlab else (if b then tlab else flab)
```

qui sera compilé vers une cascade de sauts plus courte.

```
    CJUMP(a, tlab, lab2)
lab2:
    CJUMP(b, tlab, flab)
```

Optimisations des IFs (3/3)

- Pour implémenter cette optimisation, il faut rajouter un troisième shell `Ix`.
- `Ix` nous permet de retarder la compilation des Ifs.
- Lorsque qu'un `Ix` est compilé à l'intérieur d'un autre `Ix`, on peut faire l'optimisation ci-dessus.

III

Structures de données

Traduction des tableaux (ArrayExp)

- Appel à la primitive `initArray`
- Retourne un pointeur vers un segment mémoire initialisé à la valeur par défault

Traduction des structures (RecordExp)

- Appel à la primitive `malloc` pour allouer autant de cases que nécessaire
- Compiler les `Move` qui vont bien pour initialiser la structure.