Linéarisation, réalisation des trames de piles (de ERTL à LTL, de LTL à LIN, puis de LIN à MIPS)

David Delahaye

David.Delahaye@lirmm.fr

Faculté des Sciences

Master M1 2019-2020

FACULTÉ DES SCIENCES
MONTPELLIER

Location Transfer Language (LTL)

Dans LTL, les registres physiques sont alloués

- Les fonctions (procédures) sont toujours structurées sous forme de graphe de flot de contrôle ;
- Lors du passage de ERTL à LTL, des instructions de « spill » ont été insérées, et des instructions mortes ont été supprimées en les remplaçant par des sauts inconditionnels ;
- Dans la suite, cette structure ne sera plus utile et on revient donc à une structure linéaire.

Un exemple de traduction

Fonction factorielle (en récursif)

$f(n : \text{integer}) : \text{integer}$
    if $n = 0$ then
        $f := 1$
    else
        $f := n \times f(n-1)$

Un exemple de traduction

En ERTL (rappel)

procedure $f(1)$
var $\%0, \%1, \%2, \%3, \%4, \%5, \%6$
entry $f11$
$f11$: newframe $\rightarrow f10$
$f10$: move $\%6, \$ra \rightarrow f9$
$f9$: move $\%5, \$s1 \rightarrow f8$
$f8$: move $\%4, \$s0 \rightarrow f7$
$f7$: move $\%0, \$a0 \rightarrow f6$
$f6$: li $\%1, 0 \rightarrow f5$
$f5$: blez $\%0 \rightarrow f4, f3$
$f3$: addiu $\%3, \%0, -1 \rightarrow f2$
$f2$: j $\rightarrow f20$

$f20$: move $\$a0, \%3 \rightarrow f19$
$f19$: call $f(1) \rightarrow f18$
$f18$: move $\%2, \$v0 \rightarrow f1$
$f1$: mul $\%1, \%0, \%2 \rightarrow f0$
$f0$: j $\rightarrow f17$
$f17$: move $\$v0, \%1 \rightarrow f16$
$f16$: move $\$ra, \%6 \rightarrow f15$
$f15$: move $\$s1, \%5 \rightarrow f14$
$f14$: move $\$s0, \%4 \rightarrow f13$
$f13$: delframe $\rightarrow f12$
$f12$: jr $\$ra$
$f4$: li $\%1, 1 \rightarrow f0$

Un exemple de traduction

Traduction en LTL

procedure f (1)
var 8
entry $f11$
$f11$: newframe $\rightarrow f10$
$f10$: sets local (0), $\$ra \rightarrow f9$
$f9$: j $\rightarrow f8$
$f8$: sets local (4), $\$s0 \rightarrow f7$
$f7$: move $\$s0, \$a0 \rightarrow f6$
$f6$: j $\rightarrow f5$
$f5$: blez $\$s0 \rightarrow f4, f3$
$f3$: addiu $\$a0, \$s0, -1 \rightarrow f2$
$f2$: j $\rightarrow f20$

$f20$: j $\rightarrow f19$
$f19$: call $f \rightarrow f18$
$f18$: j $\rightarrow f1$
$f1$: mul $\$v0, \$s0, \$v0 \rightarrow f0$
$f0$: j $\rightarrow f17$
$f17$: j $\rightarrow f16$
$f16$: gets $\$ra$, local(0) $\rightarrow f15$
$f15$: j $\rightarrow f14$
$f14$: gets $\$s0$, local(4) $\rightarrow f13$
$f13$: delframe $\rightarrow f12$
$f12$: jr $\$ra$
$f4$: li $\$v0, 1 \rightarrow f0$

Élimination des sauts inconditionnels

Nettoyage du graphe de flot de contrôle

- Dans LTL, l'instruction de saut inconditionnel est redondante, puisque chaque instruction mentionne explicitement son ou ses successeurs ;
- On peut donc réaliser une transformation de LTL vers lui-même qui élimine tous ces sauts inconditionnels ;
- Techniquement, il s'agit juste d'une manipulation assez triviale du graphe de flot de contrôle.

Code linéarisé (LIN)

Adieu, graphe de flot de contrôle !

- Le graphe de flot de contrôle disparaît au profit d'une suite linéaire d'instructions (on obtient un programme séquentiel) ;
- Le successeur de chaque instruction redevient implicite, sauf en cas de branchement (on saute vers une instruction avec un label) ;
- Les labels disparaissent, sauf pour les instructions cibles d'un branchement (instructions avec labels qui sont les cibles de sauts).

Un exemple de traduction

Traduction en LIN

procedure f (1)
var 8
$f11$:
newframe
sets local $(0), \$ra$
sets local $(4), \$s0$
move $\$s0, \$a0$
blez $\$s0, f4$
addiu $\$a0, \$s0, -1$
call f
mul $\$v0, \$s0, \$v0$
$f16$:
gets $\$ra$, local(0)
gets $\$s0$, local(4)
delframe
jr $\$ra$
$f4$:
li $\$v0, 1$
j $f16$

Linéarisation

Nettoyage du graphe de flot de contrôle

- La traduction de LTL vers LIN se fait par un simple parcours du graphe de flot de contrôle ;
- Lorsqu'on examine un sommet pour la première fois, on émet d'abord une étiquette, puis l'instruction associée à ce sommet, dont on examine ensuite les successeurs, en commençant par celui à qui le contrôle est transféré implicitement ;
- Lorsqu'on ré-examine un sommet déjà rencontré, on émet une instruction de saut inconditionnel vers l'étiquette correspondante ;
- On supprime a posteriori les étiquettes superflues.

Linéarisation

Variations et critères de qualité

- On peut vérifier que cet algorithme ne produit jamais de saut (conditionnel ou inconditionnel) vers un saut inconditionnel ;
- Différents ordres de parcours des sommets donnent lieu à différentes linéarisations, inverser la condition d'un saut conditionnel offre également une certaine latitude ;
- Certaines linéarisations peuvent être considérées comme préférables si elles utilisent le saut j en des points moins critiques.

Exemple de linéarisation

Deux linéarisations d'une même boucle

$\ldots$
début :
test :
(*test*)
bgtz $\$t1$, *fin*
corps :
(*corps*)
j *test*
fin :
$\ldots$

$\ldots$
début :
j *test*
corps :
(*corps*)
test :
(*test*)
blez $\$t1$, *corps*
fin :
$\ldots$

La première exécute j à chaque itération, la seconde non.

Assembleur MIPS

Gestion des trames de pile

- La gestion des trames de pile se fait par incrémentation et décrémentation explicite du registre \$sp ;
- L'accès à la pile se fait à l'aide d'un décalage fixe vis-à-vis de \$sp.

Un exemple de traduction

Traduction en MIPS

```
f17: addiu $sp, $sp, -8
     sw $ra, 4($sp)
     sw $s0, 0($sp)
     move $s0, $a0
     blez $s0, f4
     addiu $a0, $s0, -1
     jal f17
     mul $v0, $s0, $v0
f28: lw $ra, 4($sp)
     lw $s0, 0($sp)
     addiu $sp, $sp, 8
     jr $ra
f4:  li $v0, 1
     j f28
```

Organisation des trames de pile

Utilisation de $sp

- La taille des régions (paramètres entrants/sortants, variables locales) qui forment une trame de pile est enfin connue (l'allocation de registre nous a fourni cette information) ;
- Un décalage relatif à l'une des trois régions peut donc être traduit en un simple décalage vis-à-vis de $sp ;
- Les instructions newframe et delframe peuvent également être traduites en décrémentations et incrémentations de $sp.

Bonus : optimiser les appels terminaux

Appels terminaux

- Un appel g dans une fonction h est dit terminal si cet appel est la dernière opération effectuée dans h ;
- En particulier, dans ce cas là, on voit que le résultat de g devient celui de h, donc h peut « passer la main » à g ;
- Une fonction est terminale si tous ses appels de fonctions (dans son corps) sont terminaux ;
- On cherche à optimiser les appels terminaux.

Exemple : la fonction factorielle en récursif

Version non terminale

$f(n : \text{integer}) : \text{integer}$
  if $n = 0$ then
    $f := 1$
  else
    $f := n \times f(n-1)$

Exemple : la fonction factorielle en récursif

Version terminale

$f(n : \text{integer})(acc : \text{integer}) : \text{integer}$

if $n = 0$ then

$f := acc$

else

$f := f(n - 1, n \times acc)$

Appel avec $f(n, 1)$.

Optimisation des appels terminaux

Principe

- Après le retour d'un appel terminal, l'appelant se contentera de détruire sa trame puis de passer la main à son propre appelant ;
- Par conséquent, la trame de l'appelant n'a plus lieu d'être avant même l'appel, et il vaut mieux que l'appelé rende directement la main à l'appelant de l'appelant.

Optimisation des appels terminaux

Concrètement

Supposons que l'on ait un appel terminal à g dans h. Celui-ci sera compilé de la manière optimisée suivante :

- la valeur initiale des registres « callee-save » est restaurée (y compris $ra, qui contient donc l'adresse de retour dans l'appelant de h) ;
- La trame de h est désallouée ;
- Les arguments de g sont passés comme d'habitude, les quatre premiers dans les registres $a0 à $a1, les autres sur la pile ;
- Le contrôle est transféré à g par un simple saut (j au lieu de jal).

Du point de vue de l'appelé g, tout se passe comme s'il était appelé par l'appelant de h.

Exemple : la fonction factorielle en récursif

Traduction optimisée de l'appel terminal en MIPS

```
mul $a1, $a0, $a1   # calcul des arguments
addiu $a0, $a0, -1
lw $ra, 0($sp)      # restauration des callee-save
addiu $sp, 4        # désallocation de la pile
j fact              # appel de fact
```

Appels récursifs terminaux

Principe

- Dans le cas d'un appel récursif terminal, il est possible d'optimiser encore plus ;
- En effet, si f s'appelle elle-même de façon terminale, rien ne sert de restaurer les registres « callee-save », détruire la trame de f pour ensuite recréer la trame et sauvegarder les registres « callee-save » dans l'appelé ;
- Il suffit de passer le contrôle à f directement après l'allocation de la trame dans f.

Appels récursifs terminaux

Concrètement

Il convient donc de :

- Placer les arguments attendus par f, dans les registres \$a0 à \$a3 pour les premiers, et en écrasant les précédents dans la trame de l'appelant pour les autres ;
- Transférer le contrôle par un simple saut j au point situé après l'allocation de la trame et la sauvegarde des registres « callee-save ».

Exemple : la fonction factorielle en récursif

Traduction encore plus optimisée en MIPS

```
begin: addiu $sp, $sp, -4
       sw $ra, 0($sp)
loop:  blez $a0, base
       mul $a1, $a0, $a1
       addiu $a0, $a0, -1
       j loop
end:   lw $ra, 0($sp)
       addiu $sp, $sp, 4
       jr $ra
base:  move $v0, $a1
       j end
```

Exemple : la fonction factorielle en récursif

Programme équivalent

$f(n : \text{integer})(acc : \text{integer}) : \text{integer}$
  while $n > 0$ do
    $(acc := n \times acc;$
    $n := n - 1);$
  $f := acc$

Exercice

Fonction de Fibonacci

$$f(n) = \begin{cases} 0, \text{ si } n = 0 \\ 1, \text{ si } n = 1 \\ f(n-1) + f(n-2), \text{ sinon} \end{cases}$$

- Écrire la fonction de Fibonacci en itératif ;
- Écrire la fonction de Fibonacci en récursif terminal ;
- Écrire la fonction de Fibonacci en MIPS optimisé.

Exercice

Soit la fonction suivante

$$f(n) = \begin{cases} n - 10, \text{ si } n > 100 \\ f(f(n+11)), \text{ sinon} \end{cases}$$

- Que calcule cette fonction pour $n \leq 101$?
- Écrire cette fonction en récursif terminal.

Correction

Résultat

- Elle rend toujours 91 pour $n \leq 101$.

Fonction récursive terminale

$$g(n,c) = \begin{cases} n, \text{ si } c = 0 \\ g(n-10, c-1), \text{ si } n > 100 \text{ et } c \neq 0 \\ g(n+11, c+1), \text{ si } n \leq 100 \text{ et } c \neq 0 \end{cases}$$

$$f(n) = g(n,1)$$