

Production de code Assembleur

Derniers cours

- Assembleur abstrait
- Différence entre code intermédiaire et assembleur
- Filtrage de l'arbre de code intermédiaire:
 - algorithme simple
 - programmation dynamique
- contraintes posées par les différentes architectures
 - RISC: plein de registres, mais architecture load/store
 - CISC: opérations complexes, mais peu de registres

Rappel sur Assembleur MIPS

```
.data # tout ce qui suit va dans le segment DATA (donnees)
add_ : .asciiz "Testing ADD\n" # une chaine terminée par 0
fm: .asciiz "Failed test\n" # une autre chaine terminée par 0
.text # tout ce qui suit va dans le segment TEXT (code)
.globl main # Standard startup code.
main: addi $2 $0 4 # syscall 4 (print_str)
      la $a0 add_
      syscall
      addi $2 $0 1
      addi $3 $0 -1
      add $4 $0 $0
      bne $4 $0 fail
      j end
fail: addi $2 $0 4 # syscall 4 (print_str)
      la $a0 fm
      syscall
end: addiu $2 $0 10
     syscall # syscall 10 (exit)
```

Voyons ça en pratique sous l'emulateur SPIM...

Assembleur abstrait

Nous introduisons une version abstraite¹ de langage assembleur, dans laquelle on peut plonger la plupart des langages d'assemblage. Il ne s'agit plus d'une représentation arborescente: le programme est bel et bien devenu une liste d'instructions machine, définies comme suit

¹une variante du code à trois adresses en littérature

1

```
let format saytemp =
let speak(assem,dst,(src:temp list),jump) =
let saylab = Symbol.name
in let rec f = function
  ('::: 's': i::rest) ->
  (explode(saytemp(List.nth src (Char.code i - Char.code '0'))) @ f rest)
| ('::: 'd': i::rest) ->
  (explode(saytemp(List.nth dst (Char.code i - Char.code '0'))) @ f rest)
| ('::: 'j': i::rest) ->
  (explode(saylab(List.nth jump (Char.code i - Char.code '0'))) @ f rest)
| ('::: _::rest) -> ' ' :: f rest
| (c :: rest) -> (c :: f rest)
| [] -> []
in implode(f(explode assem))
in function
  OPER{o_assem=assem;o_dst=dst;o_src=src;jump=Some l}
  -> speak(assem,dst,src,l)
| OPER{o_assem=assem;o_dst=dst;o_src=src;jump=None}
  -> speak(assem,dst,src,[])
| LABEL{l_assem=assem}
  -> assem
| MOVE{m_assem=assem;m_dst=dst;m_src=src}
  -> speak(assem,{dst},{src},[])
(* end *)
```

Mon fichier assem.ml (et commentaires sur modules Ocaml)

Exemples:

```
# let t1 = OPER {o_assem="li 'd0,['s0+2]"; o_dst=[Temp.new_temp()];
                o_src=[Temp.new_temp()]; jump=None};
val t1 : Assem.instr =
  OPER
  {o_assem="li 'd0,2('s0)"; o_dst=[<abstr>]; o_src=[<abstr>]; jump=None}

# let t2 = OPER {o_assem="jal 'j0"; o_dst=[Temp.new_temp()];
                o_src=[]; jump=Some [Symbol.symbol "endloop"]};
val t2 : Assem.instr =
  OPER {o_assem="jal 'j0"; o_dst=[<abstr>]; o_src=[]; jump=Some [<abstr>]}

# format Temp.makestring t1; (* makestring sait formater un temp *)
- : string = "li t103,2(t102)"

# format Temp.makestring t2;;
- : string = "jal endloop"
```

Du code intermédiaire à l'assembleur

Une instruction assembleur peut recouvrir plusieurs noeuds d'un arbre de code intermédiaire. Par exemple,

3

```
type temp = Temp.temp (* le type des temporaires *)
type label = Temp.label (* le type des étiquettes *)
type instr = OPER of ropers (* une operation, y compris les sauts *)
            | LABEL of rlabel (* une étiquette *)
            | MOVE of rmove (* les transferts entre registres *)

and ropers = {o_assem: string; o_dst: temp list; o_src: temp list;
             jump: label list option}
and rlabel = {l_assem: string; lab: temp}
and rmove = {m_assem: string; m_dst: temp; m_src: temp}

o_assem ressemble à "bge 's0 's1 'j0" où les 's0 's1 et 'j0 sont des marqueurs pour le source numero 0, 1 dans la liste src et l'étiquette numero 0 dans la liste jump. On marque 'd0, 'd1 etc. les destinations dans dst. On fait cela pour garder un peu de structure quand même. Ici, assem est la chaîne de caractères qui décrit l'instruction assembleur tel qu'on va l'émettre, mais src est la liste des (registres) temporaires lus dans l'instruction et dest est la liste de (registres) temporaires modifiés par l'opération (oui, certaines machines modifient plus d'un registre en un seul coup). Enfin, jump est la liste d'étiquettes destination pour un saut conditionnel ou pas (sur la plupart des machines, l'étiquette pour le saut conditionnel en cas d'échec de la comparaison est juste l'étiquette de l'instruction suivante).
Ces informations supplémentaires sont utiles en cas d'analyse du programme (ex: pour l'optimisation de l'allocation des registres).
```

Mon fichier assem.ml (et commentaires sur modules Ocaml)

```
(* module Assem = struct *)

(* le module assembleur *)

type temp = Temp.temp
type label = Temp.label

type instr = OPER of ropers | LABEL of rlabel | MOVE of rmove
and ropers = {o_assem: string; o_dst: temp list; o_src: temp list;
             jump: label list option}
and rlabel = {l_assem: string; lab: label}
and rmove = {m_assem: string; m_dst: temp; m_src: temp}

(* fonctions d'utilite *)

let implode cl =
let n = List.length cl in let s = String.create n
in for i = 0 to n-1 do String.set s i (List.nth cl i) done; s

let explode s =
let n = String.length s and l = ref []
in for i = n-1 downto 0 do l := (String.get s i):: l done; !l
```

Mon fichier assem.ml (et commentaires sur modules Ocaml)

2

```
ld $5, 10($fp)
```

correspondre à l'arbre de code intermédiaire

```
MOVE(TEMP 5,
      MEM(BINOP(PLUS,
                TEMP fp,
                CONST 10)))
```

Du code intermédiaire à l'assembleur

Mais ce recouvrement n'est pas toujours unique: le même arbre de code intermédiaire

```
MOVE(TEMP 5,
      MEM(BINOP(PLUS,
                TEMP fp,
                CONST 10)))
```

peut être recouvert aussi par la séquence d'instructions assembleur

```
addi $15,$fp,10
ld $5, $15
```

ou même par

```
addi $18,$0,10 # $0 contient toujours zero sur MIPS
add $15,$fp,$18
ld $5, $15
```

Motifs et temporaires...

On appellera motif (tile en anglais signifie plutôt carreau) un fragment d'arbre de code intermédiaire qui est aussi un arbre, ayant une racine et des feuilles.

En général, une instruction assembleur correspondre à un motif, et un arbre de code intermédiaire doit pouvoir être recouvert entièrement par des motifs disjoints, qui correspondent à des instructions assembleur. Il n'est pas nécessaire de prévoir des temporaires pour les noeuds internes d'un motif (dans le cas de `ld $5, 10($fp)`, il ne nous intéresse pas de savoir par quel moyen la machine assembleur fait la somme entre FP et 10 et lit la mémoire). Par contre, on doit allouer des temporaires pour les noeuds racine (le résultat de l'instruction) et les feuilles (les paramètres de l'instruction).

Voir exemple fait au tableau pour `x:=a[10]`.

Algorithmes!

Quelle séquence d'instruction assembleur doit-on utiliser pour recouvrir un arbre de code intermédiaire?

Un critère raisonnable est la minimisation du coût de la séquence d'instructions assembleur produite (où le coût est souvent le temps d'exécution).

Vis à vis d'une notion de coût fixée, on peut obtenir des séquences assembleur:

4

optimale si on ne peut remplacer aucune instruction assembleur par une suite d'autres instructions (ayant le même effet) de coût inférieur

optimum s'il n'existe pas d'autre séquence d'instruction assembleur (ayant le même effet) de coût inférieur

Algorithme: MaximalMunch

Il est facile d'obtenir une séquence optimale:

Algorithme MaximalMunch(arbre)

Parmi tous les motifs qui peuvent recouvrir arbre en partant de la racine, choisir le plus gros.

Émettre l'instruction correspondante

Se rappeler récursivement sur les sous-arbres correspondants aux feuilles du motif choisi

Attention: cet algorithme émet les instructions dans l'ordre inverse!

Un mot sur le filtrage de motifs

Le filtrage primitif de Ocaml est très bien adapté à la sélection de motifs opérée par MaximalMunch.

Il faut mettre les motifs les plus gros *d'abord*, et le compilateur Ocaml produira un arbre de décision qui permet d'opérer le filtrage en temps linéaire (on ne reviendra pas sur une comparaison déjà effectuée).

Optimum par programmation dynamique

Il est plus complexe d'obtenir une séquence optimum:

Algorithme MoindreCout(arbre) // On suppose connus les couts // des tous les sousarbres

Parmi tous les motifs qui peuvent recouvrir arbre en partant de la racine, choisir celui tel que la somme de son coût avec les coûts des sousarbres correspondant aux feuilles est minime.

Émettre le code correspondant au motif, puis émettre le code correspondant aux sous-arbres en position de feuille pour le motif choisi.

Dans la pratique, MaximalMunch se conduit plutôt bien.

Schema d'implantation de MaximalMunch (I)

5

```
let munchstm s = instr := []; munchstm s; getresult();;
```

Mise en garde

Le coût d'un programme (en temps) n'est pas toujours exprimable facilement en terme de coût des instructions isolées, surtout dans le cadre des architectures récentes, superscalaires et pipelined.

Les algorithmes que l'on vient de voir fonctionnent dans le cas simpliste d'une architecture séquentielle traditionnelle.

Traitement des temporaires

Maintenant, il nous faut nous tourner vers le seul objet que l'on a oublié jusqu'ici: les temporaires.

Pour plein d'opérations, et maintenant pour chaque motif, il nous faut des temporaires pour contenir les résultats intermédiaires ou les paramètres de chaque instruction.

Exemple

La traduction de l'expression $1 + 2 + 3 + (4 + 5) + (6 + 7)$ ressemble à

L1:

```
addi t102 $0 6
addi t101 t102 7
addi t105 $0 4
addi t104 t105 5
addi t108 $0 1
addi t107 t108 2
addi t106 t107 3
add t103 t106 t104
add t100 t103 t101
```

Traitement simple des temporaires

Qu'est-ce qu'un temporaire?

Nous proposons ici un traitement simple, mais inefficace:

Un temporaire est traité exactement comme une variable locale d'une fonction, donc il est alloué en mémoire, sur la pile.

Donc, si dans la compilation du corps de la fonction f nous utilisons les temporaires entre $t100$ et $t114$, il y aura 14 location supplémentaires allouées sur la pile, pour le contenir, et chaque référence à un de ces temporaires se traduira par un accès en mémoire à la case correspondante.

7

```
(* on garde les instructions dans une liste *)
let instr = ref []
let emit = fun i -> instr := i::!instr; (* petite ruse *)
let getresult () = List.rev !instr; (* pour aller plus vite *)

(* abreviation pour produire des temps *)
let result(gen) = let t = Temp.new_temp() in gen t; t

let rec munchExp =
  function
  | MEM(BINOP(PLUS,e,CONST k)) ->
    result(fun r -> emit(OPER{o_assem="ld 'd0 *(string_of_int k)^(s0)" ;
      o_dst=[r]; o_src=[munchExp e]; jump=None}))
  | MEM(BINOP(PLUS,CONST k,e)) ->
    result(fun r -> emit(OPER{o_assem="ld 'd0 *(string_of_int k)^(s0)" ;
      o_dst=[r]; o_src=[munchExp e]; jump=None}))
  | MEM(CONST k) ->
    result(fun r -> emit(OPER{o_assem="ld 'd0 *(string_of_int k)^(s0)" ;
      o_dst=[r]; o_src=[]; jump=None}))
  | MEM(e) ->
    result(fun r -> emit(OPER{o_assem="ld 'd0 0('s0)" ;
      o_dst=[r]; o_src=[munchExp e]; jump=None}))
  | etcetera
```

Schema d'implantation de MaximalMunch (II)

```
and munchstm = function
  SEQ(a,b) -> (munchstm a;munchstm b)
  | MOVE(MEM(BINOP(PLUS,e1,CONST k),e2)) ->
    emit(OPER{o_assem="sw *(string_of_int k)^(s0) 's1";
      o_src=[munchExp e1; munchExp e2];o_dst=[];jump=None})
  | EXP (CALL(NAME f,args)) ->
    emit(OPER{o_assem="jal 'j0";o_src=munchArgs(0,args);
      o_dst=[] (* sur MIPS, $ra ici! *) ; jump = Some [f]})
    emit(OPER{o_assem="sub $sp $sp *(string_of_int (List.length args));
      o_src=[];o_dst=[];jump=None});
  | JUMP (NAME l,_) -> emit(OPER{o_assem="j 'j0";o_src=[];
      o_dst=[];jump=Some [l]})
  | LABEL lab -> emit(Assem.LABEL{l_assem=(Symbol.name lab)^(s0);lab=lab})
  | etcetera
```

```
and munchArgs (n,l) =
  match l with
  [] -> []
  | a::r -> let s = munchExp a in
    emit(OPER{o_assem="sw *(string_of_int n)^(s0) 's0";
      o_src=[s]; o_dst=[]; jump =None});
    s::(munchArgs (n+1,r))
```

(* la fonction qui fait tout *)

6

Stratégie naïve pour RISC et CISC

La stratégie que l'on vient de voir génère beaucoup de trafic mémoire et n'exploite pas les registres machines disponibles sur des processeurs RISC.

Sur des machines RISC et CISC, sa réalisation demande des approches légèrement différents.

Processeurs RISC

Les processeurs RISC ont beaucoup de registres, sur lesquels on peut effectuer normalement toutes les opérations.

Dans la plupart des cas, comme celui du MIPS émulé par SPIM, l'accès à la mémoire est disponible exclusivement à travers des instructions de chargement et mémorisation explicite (architecture LOAD/STORE).

Cela signifie que si on souhaite réaliser une addition entre deux temporaires, disons $t110$ et $t120$ qui se trouvent sur la pile, pour mettre le résultat dans $t115$, on ne peut pas écrire

```
add M[t115] M[t110] M[t120]
```

Processeurs RISC

Il faudra séparer la séquence en une suite "chargement, opération, mémorisation" comme dans les pseudoinstructions

```
ld $3 M[t110]
# charge dans le registre 3
# la case memoire pour t110
ld $4 M[t120]
# charge dans le registre 4
# la case memoire pour t120
add $5 $3 $4
sw $5 M[t115]
# memorise le registre 5
# dans la case memoire pour t110
```

Bien entendu, pour obtenir de l'assemble correct, il faudra remplacer les $M[t115]$ etc. par l'adresse de la case mémoire du temporaire correspondant. Typiquement, cela deviendra du $k($fp)$ où k est l'offset par rapport à $$fp$ de la case mémoire sur la pile contenant $t115$.

(suite)

Voilà donc notre approche simpliste pour un processeur RISC:

- on alloue tous les temporaires en pile
- on réserve 3 registre pour les opérations, par exemple:

8

\$10 opérande 1
\$11 opérande 2
\$12 destination

- avant de chaque opération, on charge le contenu des temporaires opérandes dans les 2 registres opérande
- on effectue l'opération
- on sauvegarde le registre résultat à l'adresse sur la pile contenant la case du temporaire destination

Stratégie simpliste pour CISC

Les processeur CISC proposent (à différence des RISC), peu de registres (6 sur le Pentium), mais sur lesquels on peut faire plein de choses (par exemple, les utiliser pour indexer la mémoire lors d'opérations arithmétiques).

Dans le cas du Pentium, on a aussi une autre restriction: les opérations sont toujours à 2 adresses et pas 3.

Pour ces processeurs, l'approche simpliste est même plus facile à mettre en place, parce que les load et store peuvent devenir implicite grâce aux modes d'adressage, et on peut se réduire à:

- on alloue tous les temporaires en pile
- on effectue une partie des opérations directement en mémoire en exploitant les adressages plus complexes.

Stratégie simpliste pour CISC (fin)

Par exemple, une pseudo-instruction à 2 opérandes du style

```
add M[t110] M[t110] M[t120]
```

pourrait devenir

```
mov edx, [ebp-12]  
add [ebp-4], edx
```

plutôt que la suite 'à la RISC'

```
mov edx, [ebp-12]  
mov eax, [ebp-4]  
add eax, edx  
mov [ebp-4], eax
```

Défauts de l'approche simpliste

Avec cette approche simpliste, et sans optimisations successives, il est fort probable que l'on retrouve des séquences de LOAD et STORE complètement inutiles, parce que la valeur chargée ou mémorisée dans ou depuis un registre est la même que la précédente.

Pour obtenir du code beaucoup plus efficace, il est nécessaire de procéder à une analyse plus fine du programme source, et à une allocation des registres machines qui ne gaspille pas des ressources.

Ceci est le but de la "liveness analysis" et de l'allocation de registres par coloriage de graphes utilisés dans les compilateurs modernes.