Génération de code

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

May 9, 2014

Objectif

- Traduire la RI en assembleur
- Le RI Tree possède encore quelques expressions difficiles à traduire en assembleur
  - Première phase: passage en forme canonique. On simplifie encore la RI
  - Deuxième phase: génération d'instructions

I

Forme canonique

Problèmes avec la RI haut-niveau (1/3)

- `CJUMP` a deux labels. Assembleur `beq .lab1` a un seul label.

```
# Ne peut pas être traduit en un seul beq.
CJUMP = a b tlab flab

cmp a b
beq tlab
b flab
```

Problèmes avec la RI haut-niveau (1/3)

- Solution: réécrire les CJUMP pour que:
  - L'étiquette `flab` soit toujours directement après le CJUMP

```
CJUMP a b tlab fakeflab
LABEL fakeflab
JUMP flab
```

Problèmes avec la RI haut-niveau (2/3)

- Les nœuds `ESEQ` font que l'ordre d'évaluation des arbres change le résultat

```
MOVE (TEMP t1) (CONST 0)
BINOP +
    ESEQ
        MOVE (TEMP t1) (CONST 42)
        TEMP t2
    TEMP t1
```

- Si on évalue d'abord la branche gauche le résultat est `42 + t2`
- Si on évalue d'abord la branche droite le résultat est `t2`
- Complique beaucoup le générateur de code

Problèmes avec la RI haut-niveau (2/3)

- Solution: réécrire l'arbre en éliminant les `ESEQ`

```
MOVE (TEMP t1) (CONST 0)
MOVE (TEMP t1) (CONST 42)
BINOP +
    TEMP t2
    TEMP t1
```

- Pour cela il faut les faire "remonter" dans l'arbre

Problèmes avec la RI haut-niveau (3/3)

- `CALL` imbriqués dans la même expression

```
BINOP (+, (CALL foo CONST 0), (CALL foo CONST 1))
```

- Les deux `CALL` retournent leur résultat dans le même registre physique.
- Problématique, car le résultat du premier `CALL` sera écrasé par le dernier `CALL` avant utilisation.

Problèmes avec la RI haut-niveau (3/3)

- Solution: réécrire l'arbre pour que les nœuds `CALL` aient pour père:
  - soit un nœud `SXP(CALL)`
  - soit un nœud `MOVE(TEMP t, CALL)`

```
MOVE
    TEMP t1
    CALL foo
        CONST 0
MOVE
    TEMP t2
    CALL foo
        CONST 1
BINOP (+) TEMP t1 TEMP t2
```

RI canonique

- Un arbre RI est **canonique** ssi il a les propriétés:
  - Les nœuds CJUMP sont toujours directement suivis par le label false.
  - Pas de nœuds `ESEQ`
  - Le père d'un `CALL` est soit un `SXP` soit un `MOVE(TEMP t, ...)`
- On utilise un ensemble de règles de réécriture

Règles de réécriture pour les ESEQ (1/3)

- On fait remonter les ESEQ en haut de l'arbre

```
(1)
ESEQ(s1, ESEQ(s2, e))        => ESEQ(SEQ(s1,s2), e)

(2)
BINOP(op, ESEQ(s,e1), e2)   => ESEQ(s, BINOP(op, e1, e2))
MEM(ESEQ(s,e1))             => ESEQ(s, MEM(e1))
CJUMP(op, ESEQ(s,e1), e2,   => SEQ(s, CJUMP(op, e1, e2,
      tlab, flab)                                tlab, flab))
```

Règles de réécriture pour les ESEQ (2/3)

```
(3)
BINOP(op, e1, ESEQ(s, e2)) => ESEQ(MOVE(TEMP t, e1),
                                   ESEQ(s, BINOP(op, e1, t)))

(4)
CJUMP(op, e1, ESEQ(s,e2),  => SEQ(MOVE(TEMP t, e1),
    tlab, flab)                 SEQ(s, CJUMP(op, t, e2,
                                             tlab, flab)))
```

- Le temporaire permet d'évaluer `e1` avant le statement `s`.
- Car `s` pourrait changer `e1`.

Règles de réécriture pour les ESEQ (3/3)

- Cas commutatif, `s` et `e1` commutent:
  - L'évaluation `s,e1` et `e1,s` est équivalente

```
(3')
BINOP(op, e1, ESEQ(s, e2)) => ESEQ(s, BINOP(op, e1, e2)))
```

```
(4')
CJUMP(op, e1, ESEQ(s,e2),  => SEQ(s, CJUMP(op, e1, e2,
    tlab, flab)                              tlab, flab)))
```

Commutativité

- On estime la commutativité de deux expressions:
  - Une constante commute avec toute autre expression
  - Le statement vide commute avec toute autre expression
- Des méthodes plus avancés existent, mais requièrent une analyse plus poussée

Élimination des ESEQ

- En appliquant récursivement les règles de réécriture on sort les ESEQ qui sont à l'intérieur d'autres expressions. Le patron général est:

```
NOEUD(e1,e2, ..., ESEQ(s, ek), ..., ek+1, ek+2, ...)
```

```
=>
```

```
SEQ (MOVE (TEMP t1, e1)
     MOVE (TEMP t2, e2)
     ...
     s
     NOEUD(t1,t2, ..., ek, ek+1, ek+2, ...)
```

Élimination des ESEQ

- En fin d'algorithme tous les ESEQ sont éliminés, car l'arbre d'une fonction est toujours un statement. Deux possibilités

```
     MOVE TEMP rv (... body ...)
ou   SXP (... body ...)

or   MOVE (TEMP rv) (ESEQ(s, e)) => SEQ(s, MOVE (TEMP rv, e)
et   SXP (ESEQ(s, e)) => SEQ(s, SXP(e))
```

- Donc récursivement tous les ESEQ sont éliminés

Réécriture des CALL

- Pour réécrire les CALL on utilise des règles similaires aux règles pour les ESEQ.

```
    BINOP(+, CALL(...), CALL(...))
=>
    MOVE(TEMP t1, CALL(...))
    MOVE(TEMP t2, CALL(...))
    BINOP(+, t1, t2)
```

Réécriture des CMOVE

- Partition en **Basic Blocks**
  - Un *basic block* est une séquence de statements qui:
    - Commence par un LABEL
    - Finit par un JUMP ou CJUMP
    - Ne contient aucun autre LABEL, JUMP ou CJUMP
  - L'exécution d'un basic block n'est donc jamais interrompue par un saut
- On décompose chaque fonction en *basic blocks*
  - L'algorithme est simple à chaque label rencontré on commence un nouveau basic block
  - à chaque JUMP/CJUMP rencontré on finit le basic block.
  - On rajoute un faux label aux basic blocks sans label au début
  - On rajoute un faux jump vers la fin de la fonction au dernier basic block

Ordonnancement des traces

- Les basics blocs peuvent être réorganisés dans un ordre quelconque sans changer la sémantique d'une fonction.
- On veut que chaque CJUMP soit suivi d'un des label cible. Sinon il faut introduire des JUMP supplémentaires:

```
CJUMP a b tlab fakeflab
LABEL fakeflab
JUMP flab
```

- Pour optimiser la génération de code on veut un ordre où:
  - chaque basic bloc est suivi par un basic block cible.

Ordonnancement des traces (Exemple)

- L'ordre 1,2,3,4 est peu efficace

```
BB1 CJUMP (BB1) (BB3)
BB2 CJUMP (END) (BB4)
BB3 JUMP  (BB2)
BB4 CJUMP (BB3) (BB2)

=>

BB1 CJUMP (BB1) (F3)
F3  JUMP (BB3)
BB2 CJUMP (END) (F4)
F4  JUMP (BB4)
BB3 JUMP (BB2)
BB4 CJUMP (BB3) (F2)
F2  JUMP (BB2)
```

Ordonnancement des traces (Exemple)

- L'ordre 1,3,2,4 est bien meilleur

```
BB1 CJUMP (BB1) (BB3)
BB3 JUMP  (BB2)
BB2 CJUMP (END) (BB4)
BB4 CJUMP (BB2) (BB3)

=>

BB1 CJUMP (BB1) (BB3)
BB3 JUMP (BB2)
BB2 CJUMP (END) (BB4)
BB4 CJUMP (BB3) (F2)
F2  JUMP (BB2)
```

Ordonnancement des traces (Exemple)

```
BB1 CJUMP (BB1) (BB3)
BB3 JUMP (BB2)              <- Ce JUMP est inutile
BB2 CJUMP (END) (BB4)       et peut-être éliminé
BB4 CJUMP (BB3) (F2)
F2  JUMP (BB2)


=>


BB1 CJUMP (BB1) (BB3)
BB3
BB2 CJUMP (END) (BB4)       et peut-être éliminé
BB4 CJUMP (BB3) (F2)
F2  JUMP (BB2)
```

Comment trouver un bon ordre

- On utilise une heuristique
  - On commence par le premier basic block de la fonction
  - On choisit juste après un des basic blocks cibles
  - On continue à enchaîner des basic blocks, jusqu'à ce que ce soit impossible
  - S'il reste encore des basic block à ordonnancer, on en choisit un et on recommence

Toujours le label False d'abord

- En assembleur le code

```
CJUMP < a b tlab flab   => CMP a,b ; BLT tlab
LABEL flab
```

s'écrit en deux instructions si le label `flab` suit immédiatement le CJUMP.

- Que faire dans le cas suivant ?

```
CJUMP < a b tlab flab
LABEL tlab
```

Toujours le label False d'abord

```
CJUMP < a b tlab flab  => CJUMP >= a b flab tlab
LABEL tlab                LABEL tlab
```

Résumé

- Pour passer en forme canonique:
  - On supprime les nœuds ESEQ
  - On réécrit les nœuds CALL
  - On réordonne les CJUMP et LABEL de manière à limiter le nombre de JUMP supplémentaires insérés.

II

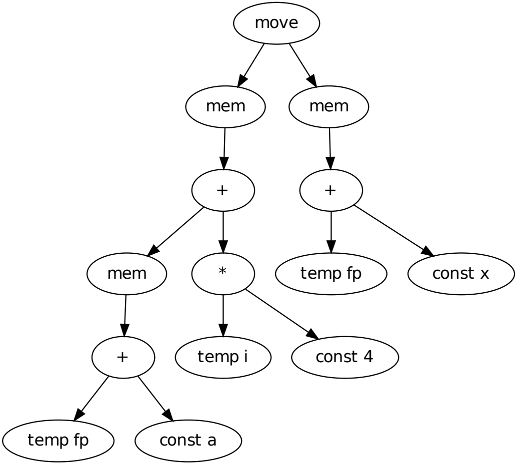
Génération de code

Objectif

- Transformer un arbre RI canonique en code assembleur ARM

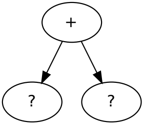
Exemple

- `a[i] := x`, a est sur la pile, x est sur la pile, i n'est pas sur la pile.

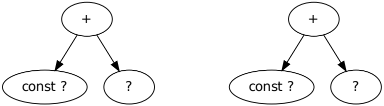


Matching d'arbre: Arithmétique

- Chaque instruction assembleur permet de traduire un sous-arbre:

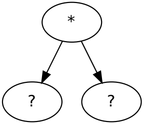


add r1, r2, r3



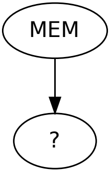
add r1, r2, #?

Matching d'arbre: Arithmétique

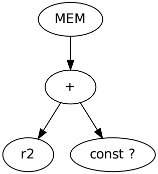


mul r1, r2, r3

Matching d'arbre: Loads



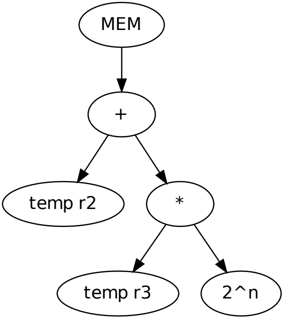
ldr r1, [r2]



ldr r1, [r2, #?]

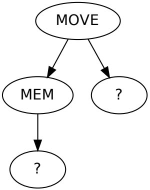
Matching d'arbre: Loads

- Les règles peuvent être très compliquées:

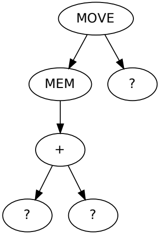


ldr r1, [r2, r3, LSL #n]

Matching d'arbre: Stores



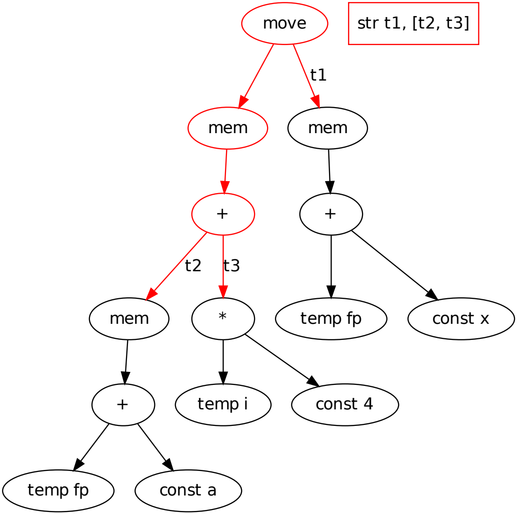
str r1, [r2]



str r1, [r2, r3]

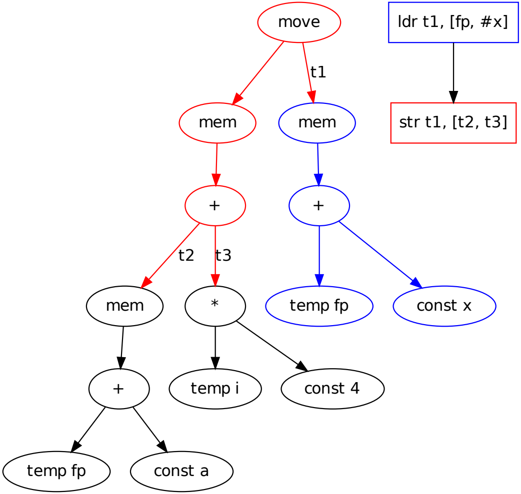
Problème: Comment recouvrir l'arbre avec les tuiles ?

- Traduction = Recouvrir l'arbre avec les tuiles élémentaires.



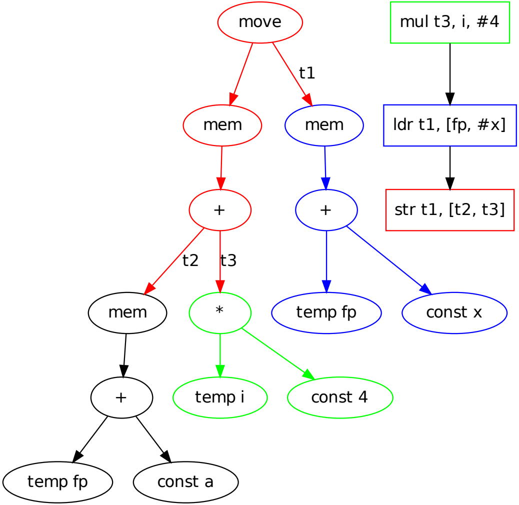
Problème: Comment recouvrir l'arbre avec les tuiles ?

- Traduction = Recouvrir l'arbre avec les tuiles élémentaires.



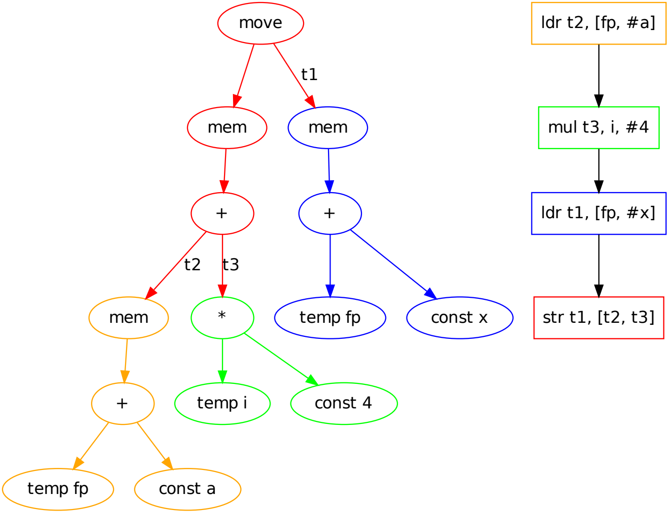
Problème: Comment recouvrir l'arbre avec les tuiles ?

- Traduction = Recouvrir l'arbre avec les tuiles élémentaires.



Problème: Comment recouvrir l'arbre avec les tuiles ?

- Traduction = Recouvrir l'arbre avec les tuiles élémentaires.



Optimalité ?

- Plusieurs recouvrements sont possibles.
- Objectif: minimiser le coût du programme final.
- On attribue à chaque tuile un coût (en cycles par exemple)
- On cherche le “tuilage” avec le coût le plus faible.

Heuristique: Maximal Munch

- Ici le coût est le nombre d'instructions.
- On suppose que chaque tuile à un coût 1
- Algorithme:
  - On cherche la tuile la plus large pour recouvrir le sommet de l'arbre
  - Cette tuile à éventuellement des sous-arbres non-couverts.
  - On itère Maximal Munch sur chaque sous-arbre.
- Maximal Munch est simple à implémenter:
  - Garantit un optimal (et non un optimum):
    - Deux tuiles voisines ne peuvent pas être remplacées par une tuile plus grosse de moindre coût.

Programmation Dynamique:

- Maximal Munch ne trouve pas nécessairement l'optimum.
- Pour trouver l'optimum global, on peut utiliser des algorithmes de programmation dynamique comme BURG.
- CW Fraser - 1992 Fast Optimal Instruction Selection and Tree Parsing

Exemple de règles JBURG

```
temp = PLUS(temp a, temp b): 1 {
    return GenHelpers.arithmetic(inst,
        "add <d>, <s>, <s>", a, b);
}

temp = PLUS(iconst a, temp b): 1 {
    return GenHelpers.arithmetic(inst,
        "add <d>, <s>, #" + a, b);
}

iconst = PLUS(iconst a, iconst b): 0 {
    return a + b;
}
```