

École Polytechnique

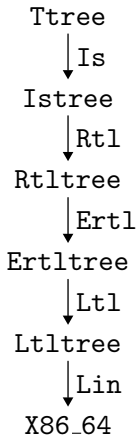
# INF564 – Compilation

Jean-Christophe Filliâtre

production de code (3/3)

la production de code optimisé a été découpée en plusieurs phases :

1. sélection d'instructions
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
  - 4.1 analyse de durée de vie
  - 4.2 construction d'un graphe d'interférence
  - 4.3 allocation de registres par coloration de graphe
5. code linéarisé (assembleur)



```
int fact(int x) {  
    if (x <= 1) return 1;  
    return x * fact(x-1);  
}
```

phase 1 : la sélection d'instructions

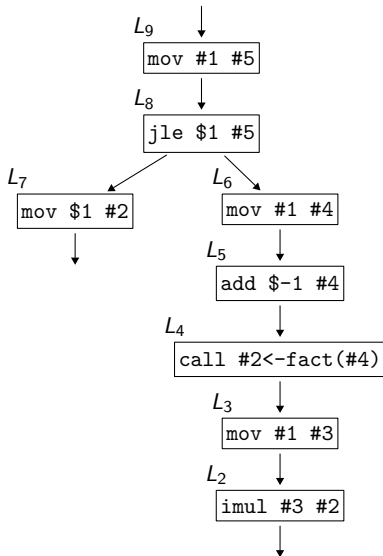
```
int fact(int x) {  
    if (setle x $1) return 1;  
    return imul x fact(addi $-1 x);  
}
```

phase 2 : RTL (*Register Transfer Language*)

```

#2 fact(#1)
  entry : L9
  exit  : L1
  locals:
  L9: mov #1 #5          --> L8
  L8: jle $1 #5           --> L7, L6
  L7: mov $1 #2           --> L1
  L6: mov #1 #4           --> L5
  L5: add $-1 #4          --> L4
  L4: #2 <- call fact(#4) --> L3
  L3: mov #1 #3           --> L2
  L2: imul #3 #2          --> L1

```



phase 3 : ERTL (*Explicit Register Transfer Language*)

```
fact(1)
  entry : L16
  locals: #6,#7
  L16: alloc_frame --> L15
  L15: mov %rbx #6 --> L14
  L14: mov %r12 #7 --> L13
  L13: mov %rdi #1 --> L9
  L9: mov #1 #5 --> L8
  L8: jle $1 #5 --> L7, L6
  L7: mov $1 #2 --> L1
  L1: goto --> L21
  L21: mov #2 %rax --> L20
```

```
L20: mov #6 %rbx --> L19
L19: mov #7 %r12 --> L18
L18: delete_frame --> L17
L17: return
L6: mov #1 #4 --> L5
L5: add $-1 #4 --> L4
L4: goto --> L12
L12: mov #4 %rdi --> L11
L11: call fact(1) --> L10
L10: mov %rax #2 --> L3
L3: mov #1 #3 --> L2
L2: imul #3 #2 --> L1
```

phase 4 : LTL (*Location Transfer Language*)

on a déjà réalisé l'**analyse de durée de vie** *i.e.* on a déterminé pour chaque variable (pseudo-registre ou registre physique) à quels moments la valeur qu'elle contient peut être utilisée dans la suite de l'exécution

```

L16: alloc_frame  --> L15  in = %r12,%rbx,%rdi  out = %r12,%rbx,%rdi
L15: mov %rbx #6   --> L14  in = %r12,%rbx,%rdi  out = #6,%r12,%rdi
L14: mov %r12 #7   --> L13  in = #6,%r12,%rdi  out = #6,#7,%rdi
L13: mov %rdi #1   --> L9   in = #6,#7,%rdi  out = #1,#6,#7
L9 : mov #1 #5     --> L8   in = #1,#6,#7    out = #1,#5,#6,#7
L8 : jle $1 #5     -> L7, L6 in = #1,#5,#6,#7 out = #1,#6,#7
L7 : mov $1 #2     --> L1   in = #6,#7      out = #2,#6,#7
L1 : goto         --> L21  in = #2,#6,#7    out = #2,#6,#7
L21: mov #2 %rax   --> L20  in = #2,#6,#7    out = #6,#7,%rax
L20: mov #6 %rbx   --> L19  in = #6,#7,%rax   out = #7,%rax,%rbx
L19: mov #7 %r12   --> L18  in = #7,%rax,%rbx out = %r12,%rax,%rbx
L18: delete_frame --> L17  in = %r12,%rax,%rbx out = %r12,%rax,%rbx
L17: return       in = %r12,%rax,%rbx out =
L6 : mov #1 #4     --> L5   in = #1,#6,#7    out = #1,#4,#6,#7
L5 : add $-1 #4    --> L4   in = #1,#4,#6,#7 out = #1,#4,#6,#7
L4 : goto         --> L12  in = #1,#4,#6,#7 out = #1,#4,#6,#7
L12: mov #4 %rdi   --> L11  in = #1,#4,#6,#7 out = #1,#6,#7,%rdi
L11: call fact(1)  --> L10  in = #1,#6,#7,%rdi out = #1,#6,#7,%rax
L10: mov %rax #2    --> L3   in = #1,#6,#7,%rax out = #1,#2,#6,#7
L3 : mov #1 #3     --> L2   in = #1,#2,#6,#7 out = #2,#3,#6,#7
L2 : imul #3 #2    --> L1   in = #2,#3,#6,#7 out = #2,#6,#7

```

on va maintenant construire un **graphe d'interférence** qui exprime les contraintes sur les emplacements possibles pour les pseudo-registres

### Définition (interférence)

*On dit que deux variables  $v_1$  et  $v_2$  **interfèrent** si elles ne peuvent pas être réalisées par le même emplacement (registre physique ou emplacement mémoire).*

comme l'interférence n'est pas décidable, on va se contenter de conditions suffisantes

soit une instruction qui **définit** une variable  $v$  : toute autre variable  $w$  vivante à la **sortie** de cette instruction peut interférer avec  $v$

cependant, dans le cas particulier d'une instruction

`mov w v`

on ne souhaite pas déclarer que  $v$  et  $w$  interfèrent car il peut être précisément intéressant de réaliser  $v$  et  $w$  par le même emplacement et d'éliminer ainsi une ou plusieurs instructions

on adopte donc la définition suivante

## Définition (graphe d'interférence)

Le **graphe d'interférence** d'une fonction est un graphe non orienté dont les sommets sont les variables de cette fonction et dont les arêtes sont de deux types : *interférence* ou *préférence*.

Pour chaque instruction qui définit une variable  $v$  et dont les variables vivantes en sortie, autres que  $v$ , sont  $w_1, \dots, w_n$ , on procède ainsi :

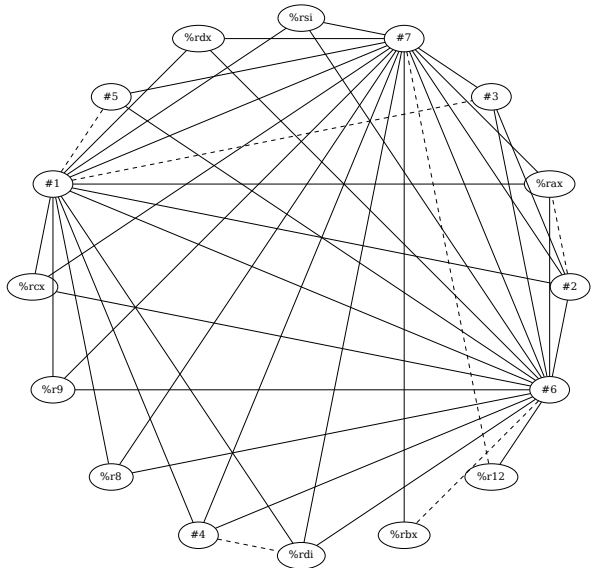
- si l'instruction n'est pas une instruction `mov w v`, on ajoute les  $n$  arêtes d'interférence  $v - w_i$
- s'il s'agit d'une instruction `mov w v`, on ajoute les arêtes d'interférence  $v - w_i$  pour tous les  $w_i$  différents de  $w$  et on ajoute l'arête de préférence  $v - w$ .

(si une arête  $v - w$  est à la fois de préférence et d'interférence, on conserve uniquement l'arête d'interférence)

voici ce que l'on  
obtient pour la  
fonction fact

9 registres physiques  
+  
7 pseudo-registres

arêtes de préférence  
en pointillés



on peut alors voir le problème de l'allocation de registres comme un problème de **coloriage de graphe** :

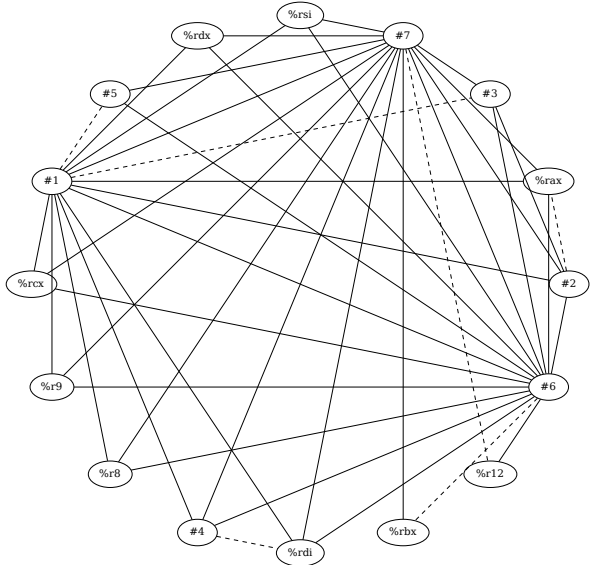
- les couleurs sont les registres physiques
- deux sommets liés par une arête d'interférence ne peuvent recevoir la même couleur
- deux sommets liés par une arête de préférence doivent recevoir la même couleur autant que possible

note : il y a dans le graphe des sommets qui sont des registres physiques, c'est-à-dire des sommets déjà coloriés

# exemple de la factorielle

observons les couleurs possibles pour les pseudo-registres

|    | couleurs possibles |
|----|--------------------|
| #1 | %r12, %rbx         |
| #2 | toutes             |
| #3 | toutes             |
| #4 | toutes             |
| #5 | toutes             |
| #6 | %rbx               |
| #7 | %r12               |



sur cet exemple, on voit tout de suite que le coloriage est impossible

- seulement deux couleurs pour colorier #1, #6 et #7
- ils interfèrent tous les trois

si un sommet ne peut être colorié, il correspondra à un emplacement de pile ; on dit qu'il est **vidé en mémoire** (en anglais on parle de *spilled register*)

quand bien même le graphe serait effectivement coloriable, le déterminer serait trop coûteux (c'est un problème NP-complet)

on va donc colorier en utilisant des **heuristiques**, avec pour objectifs

- une complexité linéaire ou quasi-linéaire
- une bonne exploitation des arêtes de préférence

l'un des meilleurs algorithmes est dû à George et Appel  
(*Iterated Register Coalescing*, 1996)

cet algorithme exploite les idées suivantes

soit  $K$  le nombre de couleurs *i.e.* le nombre de registres physiques  
(ici  $K = 9$ )

une première idée, due à Kempe (1879!), est la suivante : si un sommet a un degré  $< K$ , alors on peut le retirer du graphe, colorier le reste, et on sera ensuite assuré de pouvoir lui donner une couleur ; cette étape est appelée **simplification**

retirer un sommet diminue le degré d'autres sommets et peut donc produire de nouveaux candidats à la simplification

les sommets retirés sont donc mis sur une pile

lorsqu'il ne reste que des sommets de degré  $\geq K$ , on en choisit un comme **candidat au vidage** (*potential spill*); il est alors retiré du graphe, mis sur la pile et le processus de simplification peut reprendre

on choisit de préférence un sommet qui

- est peu utilisé (les accès à la mémoire coûtent cher)
- a un fort degré (pour favoriser de futures simplifications)

lorsque le graphe est vide, on commence le processus de coloration, appelé **sélection**

on dépile les sommets un à un et pour chacun

- s'il s'agit d'un sommet de faible degré, on est assuré de lui trouver une couleur
- s'il s'agit d'un sommet de fort degré, c'est-à-dire d'un candidat au vidage, alors
  - soit il peut être tout de même colorié car ses voisins utilisent moins de  $K$  couleurs ; on parle de **coloriage optimiste**
  - soit il ne peut être colorié et doit être effectivement vidé en mémoire (on parle d'*actual spill*)

enfin, il convient d'utiliser au mieux les arêtes de préférence

pour cela, on utilise une technique appelée **coalescence** (*coalescing*) qui consiste à fusionner deux sommets du graphe

comme cela peut augmenter le degré du sommet résultant, on ajoute un critère suffisant pour ne pas détériorer la  $K$ -colorabilité

### Définition (critère de George)

*Un sommet pseudo-registre  $v_2$  peut être fusionné avec un sommet  $v_1$ , si tout voisin de  $v_1$  qui est un registre physique ou de degré  $\geq K$  est également voisin de  $v_2$ .*

*De même, un sommet physique  $v_2$  peut être fusionné avec un sommet  $v_1$ , si tout voisin de  $v_1$  qui est un pseudo-registre ou de degré  $\geq K$  est également voisin de  $v_2$ .*

le sommet  $v_1$  est supprimé et le graphe mis à jour

s'écrit naturellement récursivement

```
simplify(g) =  
    ...  
coalesce(g) =  
    ...  
freeze(g) =  
    ...  
spill(g) =  
    ...  
select(g, v) =  
    ...
```

note : la pile des sommets à colorier est donc implicite

```
simplify(g) =  
  if il existe un sommet v sans arête de préférence  
    de degré minimal et  $< K$   
  then  
    select(g, v)  
  else  
    coalesce(g)
```

```
coalesce(g) =  
  if il existe une arête de préférence v1-v2  
    satisfaisant le critère de George  
  then  
    g <- fusionner(g, v1, v2)  
    c <- simplify(g)  
    c[v1] <- c[v2]  
    renvoyer c  
  else  
    freeze(g)
```

```
freeze(g) =  
  if il existe un sommet de degré minimal  $< K$   
  then  
    g <- oublier les arêtes de préférence de v  
    simplify(g)  
  else  
    spill(g)
```

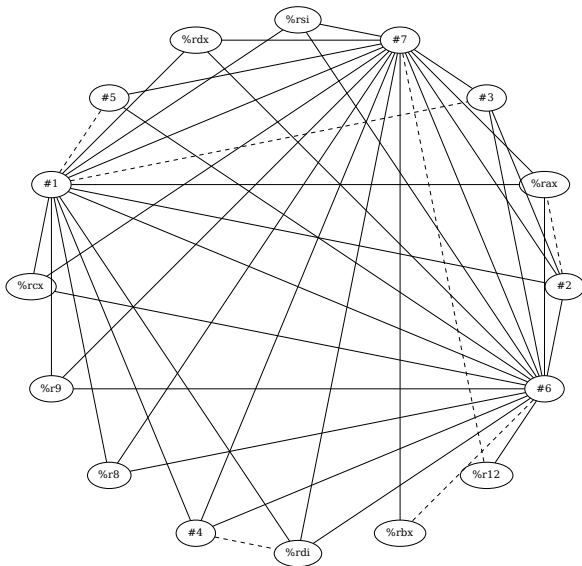
```
spill(g) =  
  if g est vide  
  then  
    renvoyer le coloriage vide  
  else  
    choisir un sommet v de coût minimal  
    select(g, v)
```

on peut prendre par exemple

$$\text{coût}(v) = \frac{\text{nombre d'utilisations de } v}{\text{degré de } v}$$

```
select(g, v) =  
  supprimer le sommet v de g  
  c <- simplify(g)  
  if il existe une couleur r possible pour v  
  then  
    c[v] <- r  
  else  
    c[v] <- spill  
  renvoyer c
```

1.  
simplify(g) →  
coalesce(g) →  
selects #3- -#1



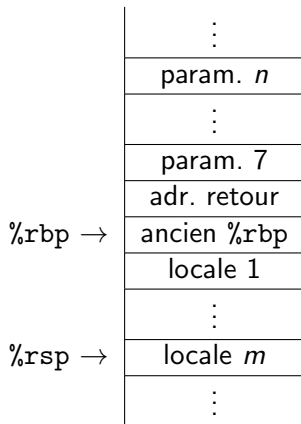
puis on dépile

|    |                     |   |                      |
|----|---------------------|---|----------------------|
| 7. | coalesce #7- - %r12 | → | c[#7] = %r12         |
| 6. | select #1           | → | c[#1] = %rbx         |
| 5. | select #6           | → | c[#6] = spill        |
| 4. | coalesce #4- - %rdi | → | c[#4] = %rdi         |
| 3. | coalesce #2- - %rax | → | c[#2] = %rax         |
| 2. | coalesce #5- - #1   | → | c[#5] = c[#1] = %rbx |
| 1. | coalesce #3- - #1   | → | c[#3] = c[#1] = %rbx |

## et les pseudo-registres vidés ?

que fait-on des pseudo-registres vidés en mémoire ?

on leur associe des emplacements sur la pile, dans la zone basse du tableau d'activation, en dessous des paramètres



plusieurs pseudo-registres peuvent occuper le même emplacement de pile, s'ils n'interfèrent pas  $\Rightarrow$  comment minimiser  $m$  ?

c'est de nouveau un problème de coloriage de graphe, mais cette fois avec une infinité de couleurs possibles, chaque couleur correspondant à un emplacement de pile différent

algorithme :

1. on fusionne toutes les arêtes de préférence (coalescence), parce que `mov` entre deux registres vidés coûte cher
2. on applique ensuite l'algorithme de simplification, en choisissant à chaque fois le sommet de degré le plus faible (heuristique)

on obtient l'allocation de registres suivante

```
#1 -> %rbx
#2 -> %rax
#3 -> %rbx
#4 -> %rdi
#5 -> %rbx
#6 -> stack -8
#7 -> %r12
```

ce qui *donnerait* le code suivant

|                        |            |                        |        |
|------------------------|------------|------------------------|--------|
| fact(1)                |            | L20: mov -8(%rbp) %rbx | --> L1 |
| entry : L16            |            | L19: mov %r12 %r12     | --> L1 |
|                        |            | L18: delete_frame      | --> L1 |
| L16: alloc_frame       | --> L15    | L17: return            |        |
| L15: mov %rbx -8(%rbp) | --> L14    | L6 : mov %rbx %rdi     | --> L5 |
| L14: mov %r12 %r12     | --> L13    | L5 : add \$-1 %rdi     | --> L4 |
| L13: mov %rdi %rbx     | --> L9     | L4 : goto              | --> L1 |
| L9 : mov %rbx %rbx     | --> L8     | L12: mov %rdi %rdi     | --> L1 |
| L8 : jle \$1 %rbx      | --> L7, L6 | L11: call fact(1)      | --> L1 |
| L7 : mov \$1 %rax      | --> L1     | L10: mov %rax %rax     | --> L3 |
| L1 : goto              | --> L21    | L3 : mov %rbx %rbx     | --> L2 |
| L21: mov %rax %rax     | --> L20    | L2 : imul %rbx %rax    | --> L1 |

comme on le constate, de nombreuses instructions de la forme

`mov v v`

peuvent être éliminées; c'était l'intérêt des arêtes de préférence

ce sera fait pendant la traduction vers LTL

on a toujours un graphe de flot de contrôle

la plupart des instructions LTL sont les mêmes que dans ERTL,  
mais les opérandes sont maintenant toutes des registres physiques ou des  
emplacement de pile

call  $f \rightarrow L$

goto  $\rightarrow L$

return

instructions identiques à celles de ERTL

load  $n(r_1) \ r_2 \rightarrow L$

store  $r_1 \ n(r_2) \rightarrow L$

instructions identiques à celles de ERTL

mais avec registres physiques

mov  $n \ d \rightarrow L$

unop  $op \ d \rightarrow L$

binop  $op \ d_1 \ d_2 \rightarrow L$

ubbranch  $br \ d \rightarrow L_1, L_2$

bbranch  $br \ d_1 \ d_2 \rightarrow L_1, L_2$

push  $d \rightarrow L$

instructions identiques à celles de ERTL

mais avec opérandes

( $d$  = registre ou emplacement de pile)

pop  $r$

nouvelle instruction

par ailleurs, `alloc_frame`, `delete_frame` et `get_param` disparaissent, au profit de manipulation explicite de `%rsp` / `%rbp`

on traduit chaque instruction ERTL en une ou plusieurs instructions LTL, en se servant

- du coloriage du graphe
- de la structure du tableau d'activation  
(qui est maintenant connue pour chaque fonction)

une variable  $x$  peut être

- déjà un registre physique
- un pseudo-registre réalisé par un registre physique
- un pseudo-registre réalisé par un emplacement de pile

dans certains cas, la traduction est facile car l'instruction assembleur permet toutes les combinaisons

exemple : l'instruction ERTL

$$L_1 : \text{mov } n \ r \rightarrow L$$

devient l'instruction LTL

$$L_1 : \text{mov } n \ \text{color}(r) \rightarrow L$$

que  $\text{color}(r)$  soit un registre physique (par ex. `movq $42, %rax`)  
ou un emplacement de pile (par ex. `movq $42, -8(%rbp)`)

dans d'autres cas, en revanche, c'est plus compliqué car toutes les opérandes ne sont pas autorisées

le cas d'un accès à la mémoire, par exemple,

$$L_1 : \text{load } n(r_1) \ r_2 \rightarrow L$$

pose un problème quand  $r_2$  est sur la pile car on ne peut pas écrire

```
movq n(r1), m(%rbp)
```

(too many memory references for 'movq')

problème similaire si  $r_1$  est sur la pile

il faut donc utiliser un registre intermédiaire

**problème** : quel registre physique utiliser ?

en adopte ici une solution simple : deux registres particuliers seront utilisés comme registres temporaires pour ces transferts avec la mémoire, et ne seront pas utilisés par ailleurs (on choisit ici `%r10` et `%r11`)

en pratique, on n'a pas nécessairement le loisir de gâcher ainsi deux registres ; on doit alors modifier le graphe d'interférence et relancer une allocation de registres pour déterminer un registre libre pour le transfert

heureusement, cela converge très rapidement en pratique (2 ou 3 étapes seulement)

avec deux registres temporaires, on peut facilement traduire toute instruction de ERTL vers LTL

exemple avec l'instruction ERTL

$$L_1: \text{load } n(r_1) \ r_2 \rightarrow L$$

|                         | $r_2$ registre physique                                                                                   | $r_2$ sur la pile                                                                                                                                                      |
|-------------------------|-----------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $r_1$ registre physique | $L_1: \text{load } n(r_1) \ r_2 \rightarrow L$                                                            | $L_1: \text{load } n(r_1) \ \%r10 \rightarrow L_2$<br>$L_2: \text{mov } \%r10 \ n_2(\%rbp) \rightarrow L$                                                              |
| $r_1$ sur la pile       | $L_1: \text{mov } n_1(\%rbp) \ \%r10 \rightarrow L_2$<br>$L_2: \text{load } n(\%r10) \ r_2 \rightarrow L$ | $L_1: \text{mov } n_1(\%rbp) \ \%r10 \rightarrow L_2$<br>$L_2: \text{load } n(\%r10) \ \%r11 \rightarrow L_3$<br>$L_3: \text{mov } \%r11 \ n_2(\%rbp) \rightarrow L_2$ |

(ici un seul registre temporaire suffirait, mais il en faut deux pour store)

on applique un traitement spécial dans certains cas

- l'instruction `mov  $r_1$   $r_2 \rightarrow L$`  est traduite par `goto  $\rightarrow L$`  lorsque  $r_1$  et  $r_2$  ont la même couleur

c'est là que l'on récolte les fruits d'une bonne allocation des registres

- l'instruction x86-64 `imul` exige que sa seconde opérande soit un registre  $\Rightarrow$  il faut utiliser un temporaire si ce n'est pas le cas
- une opération binaire ne peut avoir ses deux opérandes en mémoire  $\Rightarrow$  il faut utiliser un temporaire si ce n'est pas le cas

on peut enfin traduire `alloc_frame` et `delete_frame` en terme de manipulation de `%rsp` / `%rbp`

| ERTL                          | LTL                                                                               |
|-------------------------------|-----------------------------------------------------------------------------------|
| <code>alloc_frame → L</code>  | <code>push %rbp</code><br><code>mov %rsp %rbp</code><br><code>add -8m %rsp</code> |
| <code>delete_frame → L</code> | <code>mov %rbp %rsp</code><br><code>pop %rbp</code>                               |

|                     |                |
|---------------------|----------------|
|                     | ⋮              |
|                     | param. $n$     |
|                     | ⋮              |
|                     | param. 7       |
|                     | adr. retour    |
| $\%rbp \rightarrow$ | ancien $\%rbp$ |
|                     | locale 1       |
|                     | ⋮              |
| $\%rsp \rightarrow$ | locale $m$     |
|                     | ⋮              |

et on peut simplifier lorsque  $m = 0$

on n'a plus qu'à assembler tous les morceaux

pour traduire une fonction  $f$

1. faire l'analyse de durée de vie
2. construire le graphe d'interférence
3. le colorier
4. en déduire la valeur de  $m$
5. traduire les instructions ERTL vers LTL

pour la factorielle, on obtient le code LTL suivant

|                        |     |                        |         |
|------------------------|-----|------------------------|---------|
| fact()                 |     |                        |         |
| entry : L16            |     |                        |         |
| L16: push %rbp         | --> | L23                    |         |
| L23: mov %rsp %rbp     | --> | L22                    |         |
| L22: add \$-8 %rsp     | --> | L15                    |         |
| L15: mov %rbx -8(%rbp) | --> | L14                    |         |
| L14: goto              | --> | L13                    |         |
| L13: mov %rdi %rbx     | --> | L9                     |         |
| L9 : goto              | --> | L8                     |         |
| L8 : jle \$1 %rbx      | --> | L7, L6                 |         |
| L7 : mov \$1 %rax      | --> | L1                     |         |
| L1 : goto              | --> | L21                    |         |
| L21: goto              | --> | L20                    |         |
|                        |     |                        |         |
|                        |     | L20: mov -8(%rbp) %rbx | --> L19 |
|                        |     | L19: goto              | --> L18 |
|                        |     | L18: mov %rbp %rsp     | --> L24 |
|                        |     | L24: pop %rbp          | --> L17 |
|                        |     | L17: return            |         |
|                        |     | L6 : mov %rbx %rdi     | --> L5  |
|                        |     | L5 : add \$-1 %rdi     | --> L4  |
|                        |     | L4 : goto              | --> L12 |
|                        |     | L12: goto              | --> L11 |
|                        |     | L11: call fact         | --> L10 |
|                        |     | L10: goto              | --> L3  |
|                        |     | L3 : goto              | --> L2  |
|                        |     | L2 : imul %rbx %rax    | --> L1  |

il reste une dernière étape : le code est toujours sous la forme d'un **graphe de flot de contrôle** et l'objectif est de produire du **code assembleur linéaire**

plus précisément : les instructions de branchement de LTL contiennent

- une étiquette en cas de test positif
- une autre étiquette en cas de test négatif

alors que les instructions de branchement de l'assembleur

- contiennent une unique étiquette pour le cas positif
- poursuivent l'exécution sur l'instruction suivante en cas de test négatif

la linéarisation consiste à parcourir le graphe de flot de contrôle et à produire le code x86-64 tout en notant dans une table les étiquettes déjà visitées

lors d'un branchement, on s'efforce autant que possible de produire le code assembleur naturel si la partie du code correspondant à un test négatif n'a pas encore été visitée

dans le pire des cas, on utilise un branchement inconditionnel (`jmp`)

on utilise deux tables

- une première pour les étiquettes déjà visitées
- une seconde pour les étiquettes qui devront rester dans le code assembleur (on ne le sait pas au moment même où une instruction assembleur est produite)

la linéarisation est effectuée par deux fonctions mutuellement récursives

- une fonction `lin` produit le code à partir d'une étiquette donnée, s'il n'a pas déjà été produit, et une instruction de saut vers cette étiquette sinon
- une fonction `instr` produit le code à partir d'une étiquette et de l'instruction correspondante, sans condition

la fonction `lin` est un simple parcours de graphe

- si l'instruction n'a pas déjà été visitée, on la marque comme visitée et on appelle `instr`
- sinon on marque son étiquette comme requise dans le code assembleur et on produit un saut inconditionnel vers cette étiquette

la fonction `instr` produit effectivement le code x86-64 et rappelle récursivement `lin` sur l'étiquette suivante

|                                                                 |                                                                            |
|-----------------------------------------------------------------|----------------------------------------------------------------------------|
| $\text{instr}(L_1 : \text{mov } n \ d \rightarrow L) =$         | produire $L_1 : \text{movq } n, \ d$<br>appeler <code>lin(L)</code>        |
| $\text{instr}(L_1 : \text{load } n(r_1) \ r_2 \rightarrow L) =$ | produire $L_1 : \text{movq } n(r_1), \ r_2$<br>appeler <code>lin(L)</code> |
| <i>etc.</i>                                                     |                                                                            |

le cas intéressant est celui d'un branchement

on considère d'abord le cas favorable où le code correspondant à un test négatif ( $L_3$ ) n'a pas encore été produit

$$\begin{aligned} \text{instr}(L_1 : \text{bbranch } br \ d_1 \ d_2 \rightarrow L_2, L_3) = \\ & \text{produire } L_1 : \text{cmp } d_1, \ d_2 \\ & \text{produire } j\overline{br} \ L_2 \\ & \text{appeler } \text{lin}(L_2) \\ & \text{appeler } \text{lin}(L_3) \end{aligned}$$

sinon, il est possible que le code correspondant au test positif ( $L_2$ ) n'ait pas encore été produit et on peut alors avantageusement **inverser la condition** de branchement

$$\text{instr}(L_1 : \text{bbranch } br \ d_1 \ d_2 \rightarrow L_2, L_3) =$$

- produire  $L_1 : \text{cmp } d_1, d_2$
- produire  $jbr \ L_2$
- produire  $jmp \ L_3$

où la condition  $\overline{cc}$  est l'inverse de la condition  $cc$

enfin, dans le cas où le code correspondant aux deux branches a déjà été produit, on n'a pas d'autre choix que de produire un branchement inconditionnel

$$\text{instr}(L_1 : \text{bbranch } br \ d_1 \ d_2 \rightarrow L_2, L_3) =$$

- produire  $L_1 : \text{cmp } d_1, d_2$
- produire  $jbr \ L_2$
- produire  $jmp \ L_3$

note : on peut essayer d'estimer la condition qui sera vraie le plus souvent

le code contient de nombreux goto (boucles while dans la phase RTL, insertion de code dans la phase ERTL, suppression d'instructions mov dans la phase LTL)

on élimine ici les goto lorsque c'est possible

$$\begin{aligned} \text{instr}(L_1 : \text{goto} \rightarrow L_2) &= \text{produire jmp } L_2 \text{ si } L_2 \text{ a déjà été visité} \\ &= \text{produire l'étiquette } L_1 \\ &\quad \text{appeler lin}(L_2) \qquad \qquad \text{sinon} \end{aligned}$$

**et voilà !**

```
fact: pushq %rbp
      movq %rsp, %rbp
      addq $-8, %rsp
      movq %rbx, -8(%rbp)
      movq %rdi, %rbx
      cmpq $1, %rbx
      jle  L7
      movq %rbx, %rdi      ## useless, too bad!
      addq $-1, %rdi
      call fact
      imulq %rbx, %rax
L1:   movq -8(%rbp), %rbx
      movq %rbp, %rsp
      popq %rbp
      ret
L7:   movq $1, %rax
      jmp  L1
```

on pouvait faire un peu mieux à la main

```
fact:    cmpq    $1, %rdi        # x <= 1 ?
         jle     L3
         pushq   %rdi            # sauve x sur la pile
         decq    %rdi
         call    fact            # fact(x-1)
         popq    %rcx
         imulq   %rcx, %rax       # x * fact(x-1)
         ret

L3:
         movq    $1, %rax
         ret
```

mais il est toujours plus facile d'optimiser **un** programme

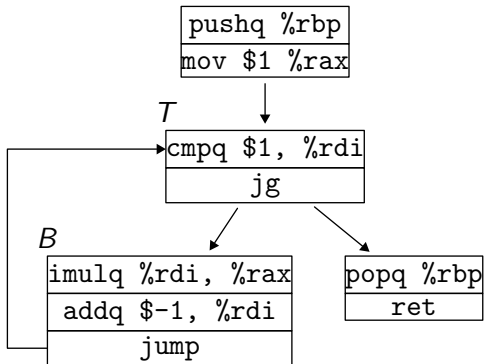
```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-2) + fib(n-1);
}
```

```
fib: pushq %rbp
     movq %rsp, %rbp
     addq $-16, %rsp
     movq %rbx, -16(%rbp)
     movq %r12, -8(%rbp)
     movq %rdi, %rbx
     cmpq $1, %rbx
     jle  L14
     movq %rbx, %rdi
     addq $-2, %rdi
     call fib
     movq %rax, %r12
```

```
     addq $-1, %rbx
     movq %rbx, %rdi
     call fib
     addq %rax, %r12
L6:  movq %r12, %rax
     movq -16(%rbp), %rbx
     movq -8(%rbp), %r12
     movq %rbp, %rsp
     popq %rbp
     ret
L14: movq %rbx, %r12
     jmp  L6
```

```
int loop(int x) {  
    int r;  
    r = 1;  
    while (x > 1) {  
        r = r * x;  
        x = x - 1;  
    }  
    return r;  
}
```

```
loop: pushq %rbp  
      movq $1, %rax  
T:    cmpq $1, %rdi  
      jg B  
      popq %rbp  
      ret  
B:    imulq %rdi, %rax  
      addq $-1, %rdi  
      jmp T
```

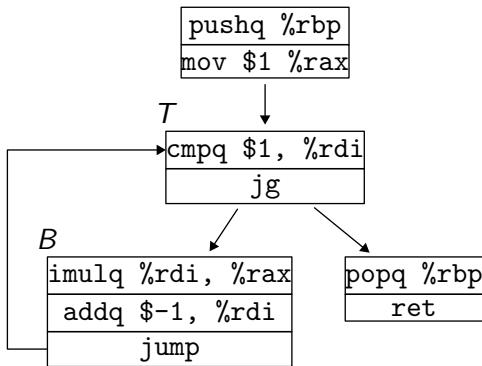


```

loop: pushq %rbp
      movq  $1, %rax
T:    cmpq  $1, %rdi
      jg   B
      popq %rbp
      ret
B:    imulq %rdi, %rax
      addq  $-1, %rdi
      jmp  T
  
```

## troisième exemple, amélioré

une meilleure linéarisation, avec un seul branchement par tour de boucle



```
loop: pushq %rbp
      movq $1, %rax
      jmp  T
B:    imulq %rdi, %rax
      addq $-1, %rdi
T:    cmpq $1, %rdi
      jg  B
      popq %rbp
      ret
```

gcc et clang le font typiquement (via un dépliage de boucle)

notre compilateur est de l'ordre de gcc -O1

|                                    | gcc -O1 | mini-C |
|------------------------------------|---------|--------|
| fib(42)                            | 3,43    | 3,67   |
| factorielle $10^9$ avec une boucle | 0,86    | 0,86   |
| 10 000 fois tak(18,12,6)           | 1,40    | 1,75   |

```
int tak(int x, int y, int z) { // Ikuo Takeuchi
    if (y < x)
        return tak(tak(x-1, y, z), tak(y-1, z, x), tak(z-1, x, y));
    return z; }
```

## **d'autres architectures**

---

- l'architecture présentée ici est celle de **CompCert**
  - les optimisations sont réalisées au niveau RTL
- le compilateur **gcc** intercale un langage SSA (cf plus loin)

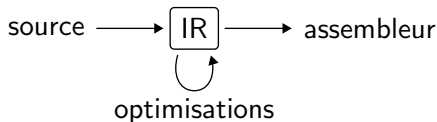
partie avant  $\rightarrow$  SSA  $\rightarrow$  RTL  $\rightarrow \dots$

et les optimisations sont réalisées au niveau SSA et au niveau RTL

- le compilateur **clang** se repose sur LLVM

il s'agit d'une infrastructure pour aider à la construction de compilateurs optimisant

LLVM propose un langage intermédiaire, IR, et des outils d'optimisation et de compilation de ce langage



le compilateur C clang est construit sur LLVM

on peut obtenir le code IR avec

```
> clang -O1 -c -emit-llvm fact.c -o fact.bc
```

et le rendre lisible avec

```
> llvm-dis fact.bc -o fact.ll
```

```

define i32 @fact(i32) {
    %2 = icmp slt i32 %0, 2
    br i1 %2, label %10, label %3
; <label>:3:                                ; preds = %1
    br label %4
; <label>:4:                                ; preds = %3, %4
    %5 = phi i32 [ %7, %4 ], [ %0, %3 ]
    %6 = phi i32 [ %8, %4 ], [ 1, %3 ]
    %7 = add nsw i32 %5, -1
    %8 = mul nsw i32 %5, %6
    %9 = icmp slt i32 %5, 3
    br i1 %9, label %10, label %4
; <label>:10:                               ; preds = %4, %1
    %11 = phi i32 [ 1, %1 ], [ %8, %4 ]
    ret i32 %11
}

```

```

define i32 @fact(i32 %x0) {
L1:
    %x2 = icmp slt i32 %x0, 2
    br i1 %x2, label %L10, label %L3
L3:
    br label %L4
L4:
    %x5 = phi i32 [ %x7, %L4 ], [ %x0, %L3 ]
    %x6 = phi i32 [ %x8, %L4 ], [ 1, %L3 ]
    %x7 = add nsw i32 %x5, -1
    %x8 = mul nsw i32 %x5, %x6
    %x9 = icmp slt i32 %x5, 3
    br i1 %x9, label %L10, label %L4
L10:
    %x11 = phi i32 [ 1, %L1 ], [ %x8, %L4 ]
    ret i32 %x11
}

```

le langage IR ressemble beaucoup à notre langage RTL

- des pseudo-registres (%2, %5, %6, etc.)
- un graphe de flot de contrôle
- des appels encore haut niveau

mais il y a aussi des différences notables

- c'est un langage typé
- le code est **en forme SSA** (*Single Static Assignment*) : chaque variable n'est affectée qu'une seule fois

bien entendu, le code d'origine est susceptible d'affecter plusieurs fois une même variable

on recourt alors à un opérateur appelé  $\Phi$  pour réconcilier plusieurs branches du flot de contrôle

ainsi,

```
%x5 = phi i32 [ %x7, %L4 ], [ %x0, %L3 ]
```

signifie que %x5 reçoit la valeur de %x7 si on vient du bloc %L4 et la valeur de %x0 si on vient du bloc %L3

l'intérêt de la forme SSA est que l'on peut maintenant

- **attacher** une propriété à chaque variable  
(par ex. valoir 42, être positif, être dans l'intervalle  $[34,55]$ , etc.)
- l'exploiter ensuite **partout** où cette variable est utilisée

la forme SSA facilite bon nombre d'optimisations

on obtient de l'assembleur avec le compilateur LLVM

```
> llc fact.bc -o fact.s
```

cette phase inclut

- l'explicitation des conventions d'appel ( $\approx$  ERTL)
- l'allocation de registres ( $\approx$  LTL)
- la linéarisation

en particulier, c'est l'allocation de registres qui va faire disparaître la majeure partie des opérations  $\Phi$  (mais quelques `mov` peuvent néanmoins être nécessaires)

```
> llc fact.bc -o fact.s
```

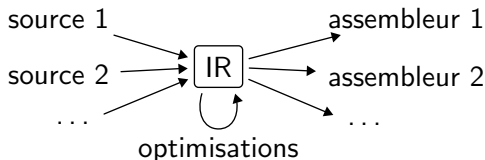
```
fact:    movl    $1, %eax
         cmpl    $2, %edi
         jl      L3
L2:      imull   %edi, %eax
         leal    -1(%rdi), %ecx
         cmpl    $2, %edi
         movl    %ecx, %edi # <- phi
         jg      2
L3:      ret
```

on peut avantageusement tirer partie de LLVM pour

- écrire un nouveau compilateur pour un langage  $S$  en se contentant d'écrire la partie avant et la traduction vers IR

et/ou

- concevoir et réaliser de nouvelles optimisations, sur le langage IR



- TD 8
  - projet mini Go
- cours 9
  - malloc et GC