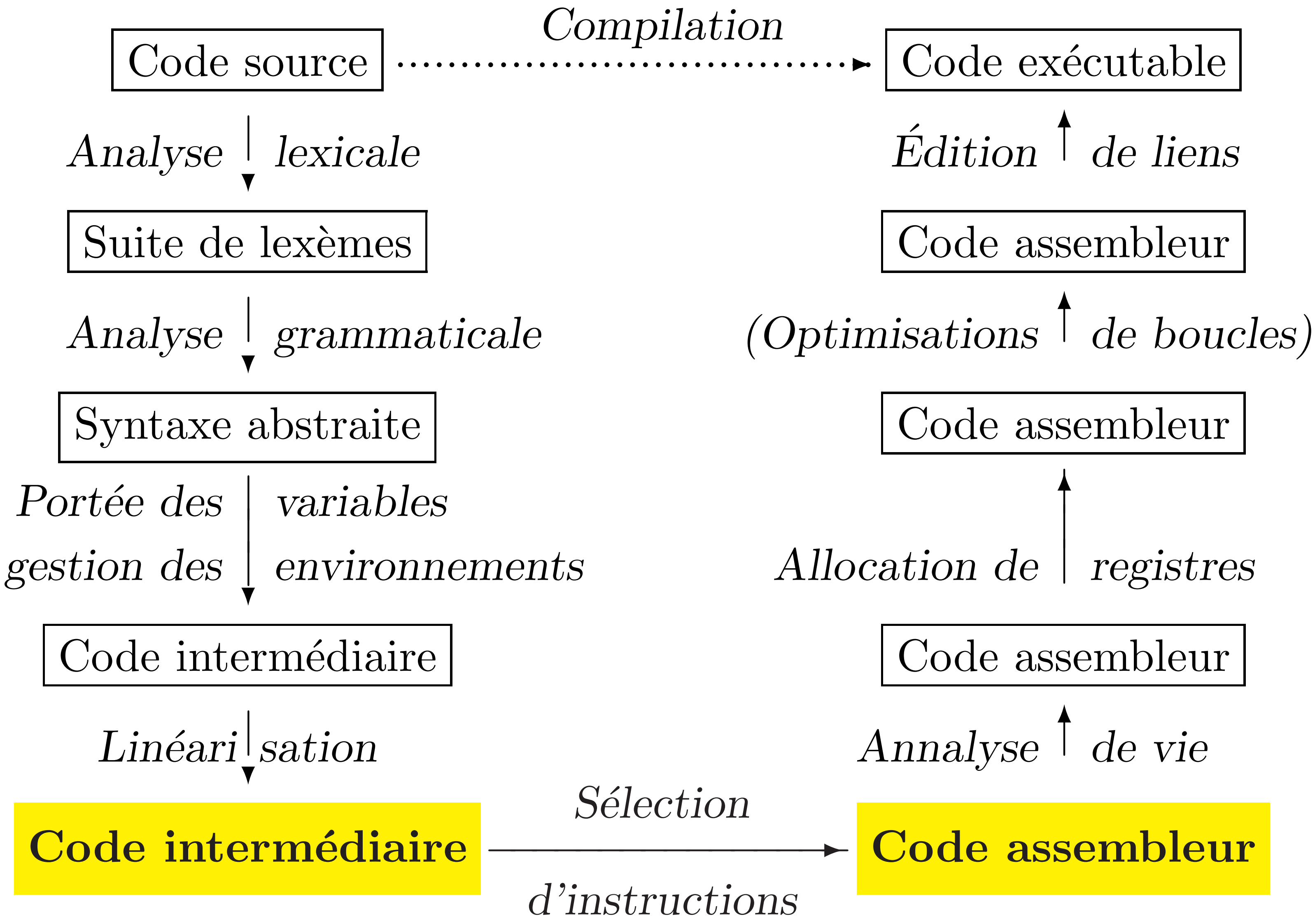
Sélection des instructions

Luc Maranget
`http://pauillac.inria.fr/~maranget`
2005


Code source
Compilation
Code exécutable
Analyse lexicale
Édition de liens
Suite de lexèmes
Code assembleur
Analyse grammaticale
(Optimisations de boucles)
Syntaxe abstraite
Code assembleur
Portée des variables
gestion des environnements
Allocation de registres
Code intermédiaire
Code assembleur
Linéarisation
Annalyse de vie
Code intermédiaire
Sélection
d'instructions
Code assembleur

La sélection

- Entièrement dépendante de la machine cible.
- Choisir les meilleures instructions possibles.
- Sans se préocuper des registres (grace aux temporaires).
- Réaliser les conventions d'appel.
- La phase la plus amusante.

Petite revue des instructions (assembleur) du MIPS

Opérations.

add r_1, r_2, r_3 # $r_1 \leftarrow r_2 + r_3$ *« trois–adresses »*
add r_1, r_2, i_{16} # $r_1 \leftarrow r_2 + i_{16}$ *immédiat*

Chargement de constantes.

li r_1, i # $r_1 \leftarrow i_{16}$ *et* $r_1 \leftarrow i_{32}$
la r_1, ℓ # $r_1 \leftarrow \ell$

Mémoire (seulement deux).

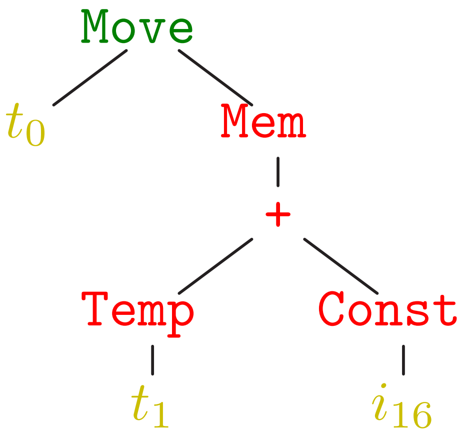
sw r_1, $i_{16}(r_2)$ # $[i_{16} + r_2] \leftarrow r_1$
lw r_1, $i_{16}(r_2)$ # $r_1 \leftarrow [i_{16} + r_2]$

Transfert entre registres.

move r_1, r_2 # $r_1 \leftarrow r_2$

Principe de base

Un parcours *postfixe* des arbres du code intermédiaire.



Mais il y a choix :

$t_2 \leftarrow i_{16}$

$t_3 \leftarrow t_1 + t_2$

$t_4 \leftarrow [0 + t_3]$

$t_0 \leftarrow t_4$

$t_3 \leftarrow t_1 + i_{16}$

$t_0 \leftarrow [0 + t_3]$

$t_0 \leftarrow [i_{16} + t_1]$

Les tuiles (*tiles*)

$t_2 \leftarrow i_{16}$

$t_3 \leftarrow t_1 + t_2$

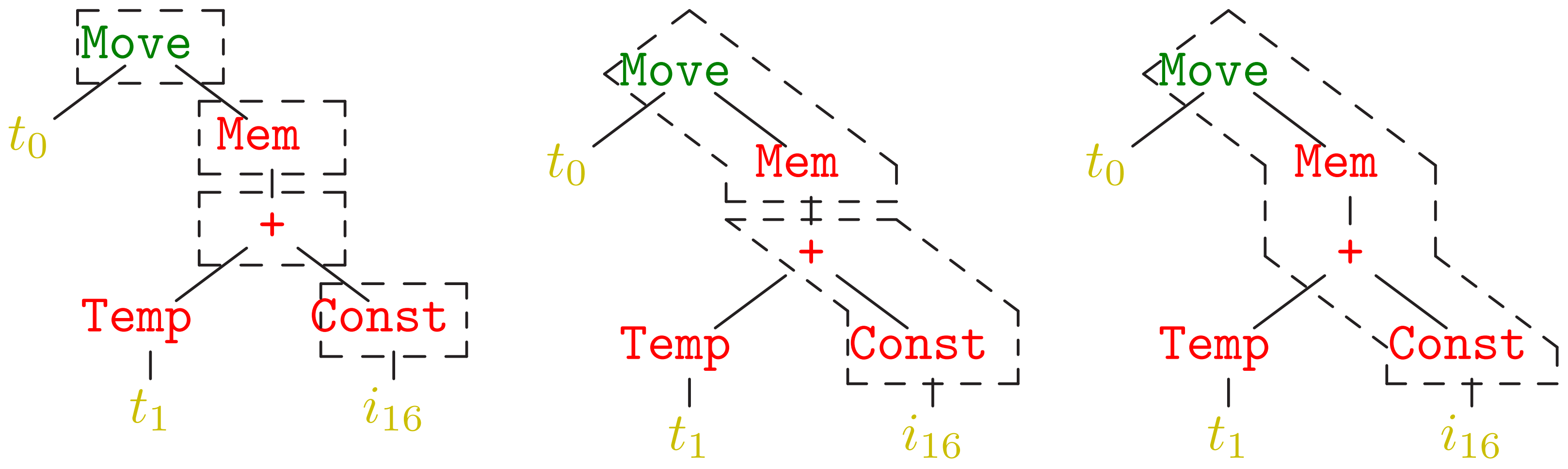
$t_4 \leftarrow [0 + t_3]$

$t_0 \leftarrow t_4$

$t_3 \leftarrow t_1 + i_{16}$

$t_0 \leftarrow [0 + t_3]$

$t_0 \leftarrow [i_{16} + t_1]$



Méthode systématique

- Associer un (deux) jeu(x) de tuiles à chaque instruction machine. Et un coût à chaque instruction machine.
- Recouvrir les arbres.
  - Recouvrement optimum : meilleur coût possible.
  - Recouvrement optimal : impossible de fusionner deux tuiles adjacentes en une seul de coût moindre.

L'optimum entraîne l'optimal, mais pas le contraîre.

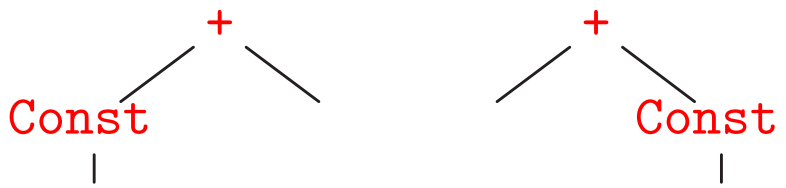
Difficultés

- Les coûts sont peu fiables (caches, *pipeline*).
- Le calcul de l'optimum est un peu difficile et coûteux (programmation dynamique, générateurs de sélecteurs).
- Et le jeu en vaut rarement la chandelle (surtout pour les RISC).

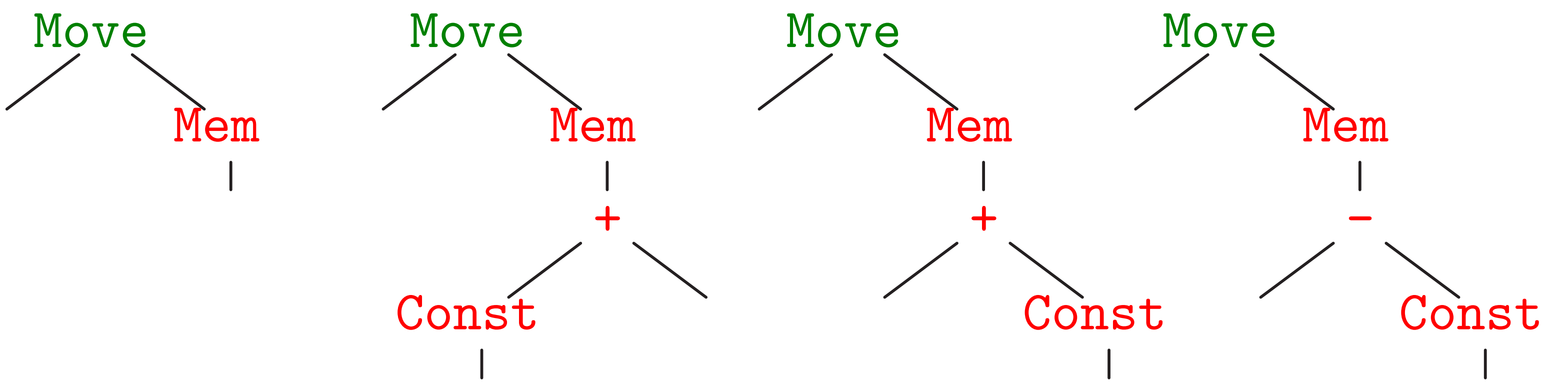
Conclusion : On approxime le coût (cycles, nombres d'instructions) et on calcule un optimal (facile en ML).

Jeu de tuiles (exemples)

Tuiles (à couvrir les expressions) de l'addition immédiate $t_1 \leftarrow t_2 + i_{16}$ (commutativité).



Tuiles (à couvrir les instructions) du *load* (accès indirect indexé) ($i_{16} = 0$, commutativité, $x - i = x + (-i)$)



Algorithme de recherche d'un optimal

À partir de la racine de l'arbre :

- Choisir la tuile de moindre coût qui recouvre l'arbre (le plus de nœuds).
- Recurser sur les sous-arbres qui dépassent.

En pratique, c'est un programme ML simple : fonction récursive qui filtre les arbres.

La différence avec l'optimum n'est pertinente que si il y a choix entre plusieures tuiles coûteuses.

L'emploi d'un générateur de sélecteur ne se justifie que pour un CISC (et encore, il faut de doute façon trouver le jeu de tuiles).

Sélecteur

```
(*  Intervalle [-2^{b-1} ... 2^{b-1}[ représentables sur b bits *)
let  seize_bits  i  = -(1 lsl 15) <= i && i < (1 lsl 15)

let emit s = Printf.printf "%s\n"

let rec emit_exp e = match e with
| Temp _  -> ()  (* cas de base, pas d'instruction *)
(* Constante entière *)
| Const i -> emit "r1 <- i" (* l'assembleur distingue r1 <- i16 et r1 <- i32 *)
(* Addition *)
| Bin (Plus, Const i, e2) when seize_bits i ->
    emit_exp e2 ; emit "r1 <- r2 + i16"
| Bin (Plus, e1, Const i) when seize_bits i ->
    emit_exp e1 ;  emit "r1 <- r2 + i16"
| Bin (Plus, e1, e2) -> emit_exp e1 ; emit_exp e2 ; emit "r1 <- r2 + r3"
```

(Accès mémoire *)*

| Mem (Bin (Plus, Const i, e2)) when seize_bits i ->

emit_exp e2 ; emit "$r_1 \leftarrow [i_{16} + r_2]$"

| Mem (Bin (Plus, e1, Const i)) when seize_bits i ->

emit_exp e1 ; emit "$r_1 \leftarrow [i_{16} + r_2]$"

| Mem (Bin (Sub , e1, Const i)) when seize_bits $(-$i$)$ ->

emit_exp e1 ; emit "$r_1 \leftarrow [i_{16} + r_2]$" *(* i_{16} est $-i$ *)*

| Mem e ->

emit_exp e ; emit "$r_1 \leftarrow [0 + r_2]$"

| ...

and emit_stm stm = **match** stm **with**

...

En pratique

Type des instructions assembleur (module `Ass`).

```
type temp = Gen.temp
type label = Gen.label

type instr =
  | Oper of string * temp list * temp list * label list option
(* Oper (mnémonique, sources, destinations, sauts) *)
  | Move of string * temp * temp
  | Label of string * label
```

La suite du compilateur ignore tout des mnémoniques (les chaînes). Seules les informations pertinentes pour l'allocation de registres sont exposées.

Sources = temporaires lus. Destinations = temporaires écrits.

Utilisation de `Ass.instr`

Les sources et les destinations sont dans des listes.

Presqu'instruction d'addition trois-adresses `add` t_1`,` t_2`,` t_3

```
Oper ("add ^d0, ^s0, ^s1", [t2 ; t3], [t1], None)
```

Instruction d'addition « immédiate » `add` t_1`,` t_2`, 20`.

```
Oper ("add ^d0, ^s0, 20", [t2], [t1], None)
```

`None` signifie : exécution en séquence.

Les sauts

Soit une étiquette ℓ de représentation externe `L123` (par `Frame.string_label`). L'instruction `b L123` est :

Oper (*"b   L123"*, [], [], Some [ℓ])

Saut conditionnel : Pour `beq` t_1`,` t_2`, L123` on a donc :

Oper (*"beq ^s0, ^s1 L123"*, [t_1 ; t_2], [], Some [ℓ ; ℓ'])

L'étiquette ℓ' est le saut en cas de condition invalidée (cf. `Cjump`)

Étiquette elle-même.

Label (*"L321:"*, ℓ')

On note le « : » suffixant l'étiquette.

Transferts entre registres

Ils sont particularisés, car on cherchera à les éliminer (en allouant t_1 et t_2 dans le même registre machine).

Move (*"move ^d0, ^s0"*, t_2, t_1)

Les registres

Les registres sont des temporaires nommés (`let`) `a0`, `sp` etc. et répartis en catégories.

```
let  arg_registers           = [a0; a1; a2; a3]
and res_registers            = [v0]
and caller_save_registers = [t0; t1; t2; t3; t4; t5; t6; t7; t8; t9]
and callee_save_registers = [ra ; s0; s1; s2; s3; s4; s5; s6; s7] (* NB ra *)

let  special_registers  =  [fp; gp; sp; zero]
let unused_registers = [at; v1; k0; k1;]
```

Émission de code à l'aide d'une « *table* »

Dans le selécteur « théorique » on affichait les instructions.
Maintenant on les accumule.

```
type 'a t

(* Créer une table *)
val create : 'a -> 'a t

(* Ajouter un element à la fin de la table *)
val emit : 'a t -> 'a -> unit

(* Vider la table dans une liste *)
val trim_to_list : 'a t -> 'a list
```

Utilisation de la table

```
(* le typage de Caml exige cet argument *)
let my_table = Table.create (Oper ("nop", [], [], None))

let emit ins = Table.emit my_table ins

let emit_move d s = emit (Move ("move ^d0, ^s0", s, d))

let memo_of_op = function
| Uplus -> "addu" (* addition non signée, pour les calculs d'adresse *)
  ...
| Ne -> "sne " (* opérations booléennes *)

let emit_op3 op d s0 s1 =
  emit (Oper (memo_of_op op^" ^d0, ^s0, ^s1"),[s0 ; s1], [d], None)
```

Le sélecteur rend un temporaire en résultat

```
let rec emit_exp e = match e with
(* Temporaire *)
| Temp t -> t
(* Constantes *)
| Const 0 -> zero (* c'est un registre *)
| Const i ->
   let d = new_temp () in
   emit (Oper ("li   ^d0, "^string_of_int i, [], [d], None) ; d
```

Le sélecteur des instructions rend rien (`() : unit`).

```
(* Opérations *)
| Bin ((Plus|Times|Uplus) as op, Const i, e2) ->
    emit_binop op e2 (Const i)
| Bin (op, e1, e2)                            ->
    emit_binop op e1 e2
| ...

and emit_binop op e1 e2 = match e2 with
| Const i when seize_bits i ->
   let s = emit_exp e1 and d = new_temp () in
   emit_op2 op d s i ; d
| _ ->
   let s0 = emit_exp e1 and s1 = emit_exp e2
   and d = new_temp () in
   emit_op3 op d s0 s1 ; d
```

Les fonctions et les conventions d'appel

- Une fonction prend ses quatre premiers arguments dans les registres `a0`, `a1`, `a2` et `a3`.
- Une fonction rend son résultat dans `v0`.
- Les registres `s0` à `s7` sont les *callee-saves*. C'est à dire que leur contenu n'est pas affecté par l'appel de fonction. Il sont sauvegardés, si necessaire, par l'appelé.
- Les registres `t0` à `t9` sont les *caller-saves*. Le contenu d'un caller-save peut être détruit par un appel de fonction. Il sont sauvegardés, si necessaire, par l'appelant.

Les appels de fonction (arguments en registres)

Il faut réaliser les conventions d'appel.

```
let emit_jal lab sources dests =
  emit (Oper ("jal "^lab, sources, dest, None))

let emit_call2 f e1 e2 =
  emit_move a0 (emit_exp e1) ; emit_move a1 (emit_exp e2) ;
  let lab = Gen.label_string (Frame.frame_name f) in
  emit_jal lab [a0; a1] (ra :: v0:: args_registers@caller_save_registers )
  let is_fun =
    match Frame.frame_result f with Some _ -> true | None -> false in
  if is_fun then Some v0 else None
```

Les arguments de l'appel de sous-routine

emit_jal lab [a0; a1] (ra :: v0 :: args_registers@caller_save_registers)

- Les sources : les arguments.
- Les destinations, `ra`, `v0` (même pour une procédure !) et tous les registres qui *peuvent* être écrits par la sous-routine.
- Pas de saut !

Voir le **jal** comme remplaçant toutes les instructions de son corps (approximation des sources et des destinations, selon les conventions d'appel).

Le corps des fonctions

Ajouter prologue (en tête) et épilogue (en queue).

```
let emit_fun f body = (* f est le frame *)
  let saved_callees = emit_prolog f in
  List.iter (fun i -> emit_stm f i) body ;
  emit_epilog f saved_callees ;
  Table.trim_to_list my_table
```

Prologue et épilogue mettent en œuvre les conventions d'appel.

Que font l'épilogue et le prologue de f ?

Prologue (étiquette `Frane.frame_name` f).

1. Allouer le frame en pile (diminuer le pointeur de pile).
2. Transférer tous les callee-saves dans des temporaires frais.
3. Copier les arguments de `a0`, `a1` etc. vers les temporaires définis pour eux lors de la génération du code intermédiaire.

Épilogue (étiquette `Frane.frame_return` f).

1. Si f n'est pas une procédure, copier le temporaire résultat dans `v0` (pendant de 3 du prologue).
2. Transférer les sauvegardes des callee-saves dans les callee-saves (repose de 2 du prologue).
3. Libérer le frame (repose de 1 du prologue).
4. Revenir de la fonction (repose de l'instruction d'appel).

La taille du frame n'est pas connue

Solution : une constante symbolique de l'assembleur (e.g. `Gen.string_label (Frame.frame_name` f`)^"_size"`)

```
    f_size  = 12                # mis là après l'allocation de registres

# Code produit par le sélecteur
f:  # prologue de f
    subu $sp, $sp, f_size
    ...

f_end:  # épilogue de f
    ...
    addu $sp, $sp, f_fize
    j    $ra
```

Réalisation de la convention *callee-save*

Les sauvegardes des callee-saves sont des temporaires :

- Ils sont insensibles aux appels de fonctions (sémantique).
- Le code de f n'y touchera pas (ils sont frais).

Donc, il contiennent bien les valeurs des callee-saves à l'entrée de l'épilogue.

On aura (si deux callee-saves `ra` et `s0`).

```
f: # prologue de f
    move $ra, $111
    move $s0, $112
    ...
f_end: # épilogue de f
    move $111, $ra
    move $112, $s0
```

En pratique, deux cas visés, sauvegarde allouée dans le callee-save lui-même ou en pile.

Sources et destinations de l'instruction de retour

Les voici :

emit (Oper *"j $ra"*, v0:: callee_save_registers , [], Some [])

- Les sources : `ra` (normal, explique `ra` callee-save), `v0` (sauf pour une procédure), les callee-saves.
- Les destinations, rien.
- Les sauts. Il y a saut, mais vers l'inconnu (sortie).

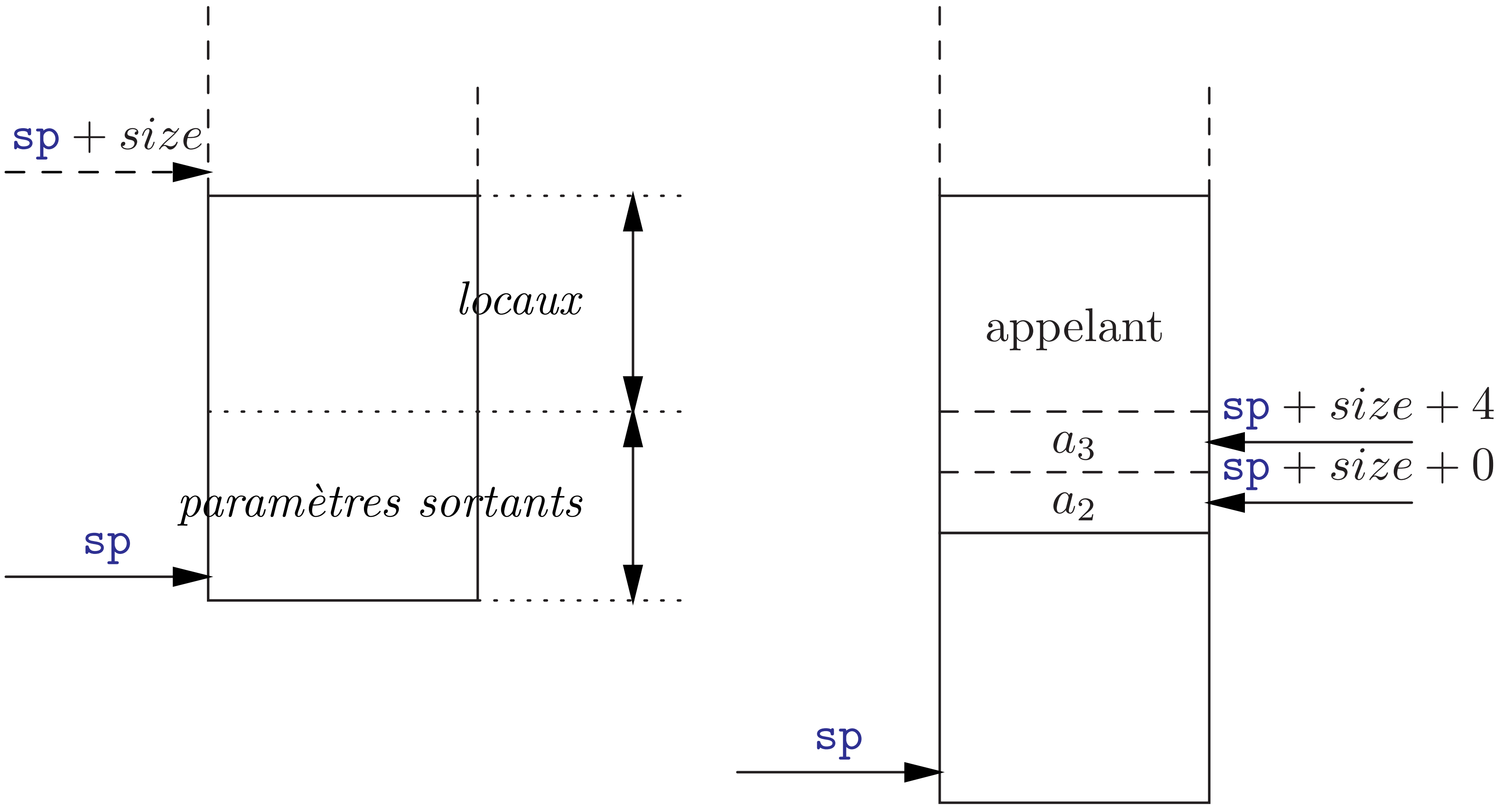
Ici encore voir l'instruction **j $ra** comme approximant toutes les instructions qui peuvent suivre.

Exemple, épilogue d'une fonction à deux arguments

```
let emit_prolog_2 f =
  let f_size = Gen.label_string (Frame.frame_label f)^"_size" in
(* point d'entrée et allocation du frame *)
  emit_label (Frame.frame_label f) ;
  emit (Oper ("subu $sp, $sp, "^f_size, [], [], None)) ;
(* sauvegarde des callee-saves *)
  let saved_ra = new_temp () and saved_s0 = new_temp () ;
  emit_move saved_ra ra ; emit_move saved_s0 s0 ;
(* récupérer les arguments *)
  let [t1 ; t2] = Frame.frame_args f in
  emit_move t1 a0 ;
  emit_move t2 a1 ;
(* rendre les sauvegardes des callee-saves, pour l'épilogue *)
  [saved_ra ; saved_s0]
```

Passage des arguments en pile

Impact sur l'appel (allocation du sommet du frame) et sur le prologue (aller chercher les arguments sur la pile)



Un source

```
program
var x : integer ;
function fact (n : integer) : integer;
begin
  if n <= 1 then fact := 1
  else fact := n * fact (n - 1);
end ;

begin
   read (x);
   writeln (fact (x))
end.
```

Le code intermédiaire (toutes optimisations faites)

```
function fact
args =  $t108, result = $t107

  Cjump L12 L13 (<= $t108 1)
L13:
  (set $t109
    (call fact (- $t108 1)))
  (set $t107 (* $t108 $t109))
  Jump fact_end
L12:
  (set $t107 1)
```

```
procedure main
  (set $t110 (call read_int))
  (setmem $t28 $t110)
  (set $t111 (call fact (mem $t28)))
  Exp (call println_int $t111)
```

Après sélection

```
fact :
      subu $sp, $sp, fact_f
      move $112, $ra
      move $113, $s0
      move $108, $a0
      li   $114, 1
      ble  $108, $114, L12
L13:
      sub  $a0, $108, 1
      jal  fact
      move $109, $v0
      mul  $107, $108, $109
      b    fact_end
```

```
L12:
      li   $107, 1
fact_end:
      move $v0, $107
      move $ra, $112
      move $s0, $113
      addu $sp, $sp, fact_f
      j    $ra
```

Code final possible

```
    fact_f=8
fact:
    subu $sp, $sp, fact_f
    sw   $ra, 0($sp) # store $112
    sw   $s0, 4($sp) # store $113
    move $s0, $a0
    li   $v0, 1
    ble  $s0, $v0, L12
L13:
    sub  $a0, $s0, 1
    jal  fact
    mul  $v0, $s0, $v0
    b    fact_end
```

```
L12:
    li   $v0, 1
fact_end:
    lw   $ra, 0($sp) # load $112
    lw   $s0, 4($sp) # load $113
    addu $sp, $sp, fact_f
    j    $ra
```