École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 10 / 10 décembre 2009

Objectif

l'objectif de ce cours et du prochain : produire du **code efficace**

jusqu'ici, nous avons très mal utilisé les capacités de l'assembleur MIPS :

- très peu de registres utilisés, alors qu'il y en a 32
  - arguments et variables locales systématiquement sur la pile
  - calculs intermédiaires sur la pile également
- instructions mal utilisées
  - exemple : on n'a jamais utilisé `addi` qui permet d'ajouter une constante

```
        addi  $t0, $t1, 12
```

(on utilisait `li` et un temporaire de plus, parfois stocké en pile !)

Phases du compilateur

il est illusoire de chercher à produire du code efficace en une seule passe

la production de code va être décomposée en **plusieurs phases**

1. sélection d'instructions
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
5. code linéarisé
6. assembleur

le point de départ est l'arbre de syntaxe abstraite issu du typage

Orthogonalité

cette architecture de compilateur est valable pour tous les grands paradigmes de programmation (impérative, fonctionnelle, objet, etc.)

pour l'illustrer, on fait néanmoins le choix d'un langage particulier, en l'occurrence un petit fragment du langage C

Mini-C

on considère un fragment très simple du langage **C** avec

- des entiers
- des structures allouées sur le tas avec `malloc` (uniquement des pointeurs sur ces structures et pas d'arithmétique de pointeurs)
- des fonctions
- une « primitive » pour afficher un entier : `printf("%d\n",`e`)`

Mini-C

$$
\begin{array}{rcl}
E & \to & n \\
& | & L \\
& | & L = E \\
& | & E \ op \ E \mid \texttt{-}\ E \mid \texttt{!}\ E \\
& | & x(E, \ldots, E) \\
& | & \texttt{malloc(sizeof(struct}\ x\texttt{))} \\
L & \to & x \\
& | & E\texttt{->}x \\
op & \to & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
& | & \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
D & \to & V \\
& | & T\ x(T\ x, \ldots, T\ x)\ B \\
& | & \texttt{struct}\ x\ \{V \ldots V\}\texttt{;}
\end{array}
\qquad
\begin{array}{rcl}
S & \to & E\texttt{;} \\
& | & \texttt{if (}E\texttt{)}\ S \\
& | & \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
& | & \texttt{while (}E\texttt{)}\ S \\
& | & \texttt{return}\ E\texttt{;} \\
& | & \texttt{printf("\%d\textbackslash n",}E\texttt{);} \\
& | & B \\
B & \to & \{\ V \ldots V\ S \ldots S\ \} \\
V & \to & T\ x, \ldots, x\texttt{;} \\
T & \to & \texttt{int} \mid \texttt{struct}\ x\ \texttt{*} \\
P & \to & D \ldots D
\end{array}
$$

Exemples

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

```
struct L { int val; struct L * next; };

int print(struct L * l) {
  while (l != 0) {
    printf("%d\n",l->val);
    l = l->next;
  }
  return 0;
}
```


Point de départ

on suppose l'analyse sémantique effectuée ; l'arbre résultant est le suivant :

```
type file = { gvars : decl_var list; funs : decl_fun list; }
and decl_var = typ × ident
and decl_fun = {
  fun_typ : typ; fun_name : ident;
  fun_formals : decl_var list; fun_body : block; }
and block = decl_var list × stmt list
and stmt =
  | Sskip
  | Sexpr of expr
  | Sif of expr × stmt × stmt
  | Swhile of expr × stmt
  | Sblock of block
  | Sreturn of expr
  | Sprintf of expr
```

Point de départ

dans les expressions, on a déjà distingué variables locales et globales, et les noms sont uniques

```
and expr = { expr_node : expr_node; expr_typ  : typ }
and expr_node =
  | Econst of int
  | Eaccess_local of ident
  | Eassign_local of ident × expr
  | Eaccess_global of ident
  | Eassign_global of ident × expr
  | Eaccess_field of expr × int       (* indice du champ *)
  | Eassign_field of expr × int × expr
  | Eunop of unop × expr                          (* - ! *)
  | Ebinop of binop × expr × expr  (* + - * / == > etc. *)
  | Ecall of ident × expr list
  | Emalloc of structure
```


Phase 1 : sélection d'instructions

la première phase est la **sélection d'instructions**

objectif :

- remplacer les opérations arithmétiques du C par celles de MIPS
- remplacer les accès aux champs de structures par les opérations `lw` et `sw` de MIPS

Opérations arithmétiques

on peut naïvement traduire chaque opération arithmétique de C par l'instruction correspondante de MIPS

cependant, MIPS fournit des instructions permettant une plus grande efficacité, notamment

- addition d'un registre et d'une constante 16 bits signée
- décalage des bits vers la gauche ou la droite, correspondant à une multiplication ou à une division par une puissance de deux
- comparaison d'un registre avec une constante 16 bits signée

Évaluation partielle

d'autre part, il est souhaitable d'évaluer autant d'expressions que possible pendant la compilation (évaluation partielle)

exemples : on peut simplifier

- $(1 + e_1) + (2 + e_2)$ en $e_1 + e_2 + 3$
- $e + 1 < 10$ en $e < 9$
- $!(e_1 < e_2)$ en $e_1 \geq e_2$
- $0 \times e$ en 0, mais seulement si e est **pure** *i.e.* sans effet de bord

Nouvelle syntaxe abstraite

on se donne de nouveaux arbres pour le résultat de la sélection d'instructions

les opérations sont maintenant celles de MIPS

```
type munop = Mmove | Maddi of int | Mseqi of int | ...
type mbinop = Madd | Msub | Mmul | Mdiv | Meq | Mneq | Mlt ...
```

et elles remplacent les opérations du C

```
type expr =
  | Emunop of munop × expr              (* remplace Eunop *)
  | Embinop of mbinop × expr × expr     (* remplace Ebinop *)
  | ...
```

on conserve cependant && et || pour l'instant

```
  | Eand of expr × expr
  | Eor of expr × expr
```


Des constructeurs « intelligents »

pour réaliser l'évaluation partielle, on va utiliser des *smart constructors*

plutôt que d'écrire `Embinop (Madd, e1, e2)` directement, on introduit une fonction

```
val mk_add : expr → expr → expr
```

qui effectue d'éventuelles simplifications et se comporte comme le constructeur sinon

Le constructeur de l'addition

voici quelques simplifications possible pour l'addition

```
let rec mk_add e1 e2 = match e1, e2 with
  | Econst n1, Econst n2 →
      Econst (n1 + n2)
  | e, Econst 0 | Econst 0, e →
      e
  | Emunop (Maddi n1, e), Econst n2
  | Econst n2, Emunop (Maddi n1, e) →
      mk_add (Econst (n1 + n2)) e
  | e, Econst n | Econst n, e when is16op n →
      Emunop (Maddi n, e)
  | _ →
      Embinop (Madd, e1, e2)
```

avec

```
let is16op n = -32768 ≤ n && n ≤ 32767
```


Terminaison et Correction

deux aspects sont essentiels concernant ces simplifications

- la sémantique des programmes doit être préservée
  - exemple : si un ordre d'évaluation gauche/droite est spécifié, on ne peut pas simplifier $(0 - e_1) + e_2$ en $e_2 - e_1$ si e_1 ou e_2 n'est pas pure
- la fonction de simplification doit terminer
  - il faut trouver une grandeur positive sur l'expression simplifiée qui diminue strictement à chaque appel récursif du *smart constructor*

Traduction

la traduction se fait alors mot à mot

```
let rec expr e = match e.Ttree.expr_node with
  | Ttree.Ebinop (Badd, e1, e2) →
      mk_add (expr e1) (expr e2)
  | Ttree.Ebinop (Bsub, e1, e2) →
      mk_sub (expr e1) (expr e2)
  | Ttree.Eunop (Ptree.Unot, e) →
      mk_not (expr e)
  | Ttree.Eunop (Ptree.Uminus, e) →
      mk_sub (Econst 0) (expr e)
  | ...
```

et c'est un morphisme pour les autres constructions (`Eaccess_local`, `Eaccess_global`, `Ecall`, etc.)

Accès à la mémoire

la sélection d'instruction introduit également des opérations explicites d'accès à la mémoire

une adresse mémoire est donnée par une expression et un décalage sur 16 bits signé (pour tirer parti de l'adressage relatif du MIPS)

```
type expr =
  | ...
  | Eload of expr × int
  | Estore of expr × int × expr
```


Accès à la mémoire

dans notre cas, ce sont les accès aux champs de structures qui sont transformés en accès à la mémoire

on est ici dans un cas simple où chaque champ occupe exactement un mot (et on suppose de plus que chaque structure a moins de 8192 champs)

d'où

```
let rec expr e = match e.Ttree.expr_node with
  | ...
  | Ttree.Eaccess_field (e, n) →
      Eload (expr e, n × word_size)
  | Ttree.Eassign_field (e1, n, e2) →
      Estore (expr e1, n × word_size, expr e2)
```

avec ici

```
let word_size = 4
```


Pour le reste

pour le reste, rien à signaler (la sélection d'instructions est un morphisme en ce qui concerne les instructions du C)

on en profite cependant pour oublier les types et regrouper l'ensemble des variables locales au niveau de la fonction

```
type deffun = {
  fun_name : ident;
  fun_formals : string list;
  fun_locals : string list;
  fun_body : stmt list;
}

type file = {
  gvars : string list;
  funs : deffun list;
}
```

Phase 2 : RTL

la deuxième phase est la transformation vers le langage **RTL** (*Register Transfer Language*)

objectif :

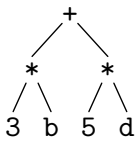
- détruire la structure arborescente des expressions et des instructions au profit d'un **graphe de flot de contrôle**, qui facilitera les phases ultérieures ; on supprime en particulier la distinction entre expressions et instructions
- introduire des **pseudo-registres** pour représenter les calculs intermédiaires ; ces pseudo-registres sont en nombre illimité et deviendront plus tard soit des registres MIPS, soit des emplacements de pile

Exemple

considérons l'expression C

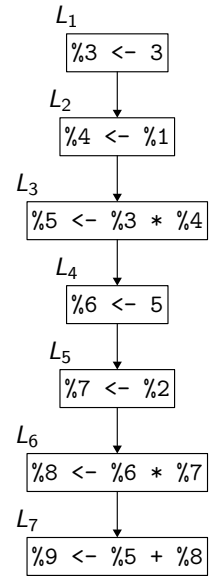
```
3*b + 5*d
```

c'est-à-dire l'arbre



supposons que `b` est dans le pseudo-registre `%1` et `d` dans le pseudo-registre `%2`

alors on construit le graphe suivant :



Pseudo-registres

on se donne un module `Register` pour les pseudo-registres

```
type t

val fresh : unit → t

module S : Set.S with type elt = t
```


Représentation du graphe de flot de contrôle

on se donne également un module `Label` pour des étiquettes représentant les sommets du graphe de flot de contrôle

```
type t

val fresh : unit → t

module M : Map.S with type key = t
```

un graphe est alors simplement un dictionnaire associant une instruction RTL à chaque étiquette

```
type graph = instr Label.M.t
```

Les instructions de RTL

inversement, chaque instruction RTL indique quelle est l'étiquette suivante (ou les étiquettes suivantes) ; ainsi

```
type instr =
  | Econst of register × int × label
  | ...
```

l'instruction `Econst (r, n, l)` signifie « charger la valeur *n* dans le pseudo-registre *r* et transférer le contrôle à l'étiquette *l* »

Autres instructions

de même, on trouve l'accès aux variables globales (toujours représentées par des identificateurs), les accès à la mémoire, `malloc` et `printf`

```
type instr =
  ...
  | Eaccess_global of register × ident × label
  | Eassign_global of register × ident × label

  | Eload of register × register × int × label
  | Estore of register × register × int × label

  | Emalloc of register × int × label
  | Eprintf of register × label
```


Opérations arithmétiques

enfin, les opérations arithmétiques manipulent maintenant des pseudo-registres

```
type instr =
  ...
  | Emunop of
      register × munop × register × label
  | Embinop of
      register × mbinop × register × register × label
```


Construction du graphe

pour construire le graphe de flot de contrôle, on le stocke (temporairement) dans une référence

```
let graph = ref Label.M.empty
```

et on se donne une fonction pour ajouter une arête dans le graphe

```
let generate i =
  let l = Label.fresh () in
  graph := Label.M.add l i !graph;
  l
```

Traduction des expressions

on traduit les expressions grâce à une fonction

```
val expr : register → expr → label → label
```

qui prend en arguments

- le registre de destination de la valeur de l'expression
- l'expression à traduire
- l'étiquette de sortie *i.e.* correspondant à la suite du calcul

et renvoie l'étiquette d'entrée du calcul de cette expression

on construit donc le graphe en partant de la fin de chaque fonction

Traduction des expressions

la traduction est relativement aisée

```
let rec expr destr e destl = match e with
  | Istree.Econst n →
      generate (Econst (destr, n, destl))
```

lorsque nécessaire, on introduit des pseudo-registres frais

```
  | Istree.Embinop (op, e1, e2) →
      let tmp1 = Register.fresh () in
      let tmp2 = Register.fresh () in
      expr tmp1 e1
        (expr tmp2 e2
          (generate (Embinop (destr, op, tmp1, tmp2, destl))))
```


Traduction des expressions

pour les variables locales, on se donne une table indiquant quel pseudo-registre est associé à chaque variable

```
  | Istree.Eaccess_local x →
      let rx = Hashtbl.find locals x in
      generate (Emunop (destr, Mmove, rx, destl))
```

où `Mmove` est l'opération unaire de MIPS move

etc.

Instructions

pour traduire les opérateurs `&&` et `||` et les instructions `if` et `while` il nous faut des instructions RTL de **branchement**

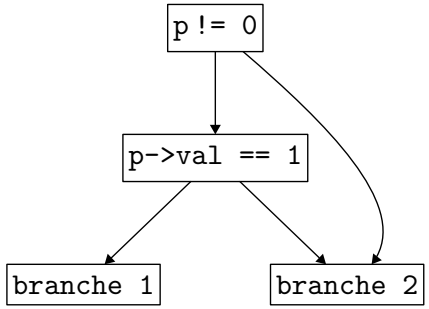
```
type instr =
  ...
  | Emubranch of mubranch × register × label × label
  | Embbranch of
      mbbranch × register × register × label × label
  | Egoto of label
```

avec

```
type mubranch = Mbeqz | ...

type mbbranch = Mblt | Mble | ...
```

Exemple

```
if (p != 0 && p->val == 1)
  ...branche 1...
else
  ...branche 2...
```



(les quatres blocs sont schématiques ; ils se décomposent en réalité en sous-graphes)

Traduction d'une condition

pour traduire une condition, on définit une fonction

```
val condition : expr → label → label → label
```

les deux étiquettes passées en arguments correpondent à la suite du calcul dans les cas où la condition est respectivement vraie et fausse

on renvoie l'étiquette d'entrée de l'évaluation de la condition

Traduction d'une condition

```
let rec condition e truel falsel = match e with
  | Istree.Eand (e1, e2) →
      condition e1 (condition e2 truel falsel) falsel
  | Istree.Eor (e1, e2) →
      condition e1 truel (condition e2 truel falsel)
  | Istree.Embinop (Mle, e1, e2) →
      let tmp1 = Register.fresh () in
      let tmp2 = Register.fresh () in
      expr tmp1 e1
      (expr tmp2 e2
      (generate (Embbranch (Mble, tmp1, tmp2, truel, falsel))
  | e →
      let tmp = Register.fresh () in
      expr tmp e
      (generate (Emubranch (Mbeqz, tmp, falsel, truel)))
```

(on peut bien entendu traiter plus de cas particuliers)

Traduction des instructions

pour traduire `return`, on se donne un pseudo-registre `retr` pour recevoir le résultat de la fonction et une étiquette `exitl` correspondant à la sortie de la fonction ; sinon, on se donne une étiquette `destl` correspondant à la suite du calcul

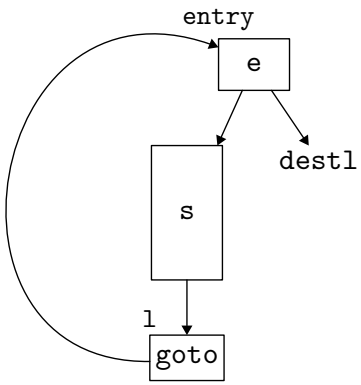
```
let rec stmt retr s exitl destl = match s with
  | Istree.Sskip →
      destl

  | Istree.Sreturn e →
      expr retr e exitl

  | Istree.Sif (e, s1, s2) →
      condition e
        (stmt retr s1 exitl destl)
        (stmt retr s2 exitl destl)
```

etc.

Boucle

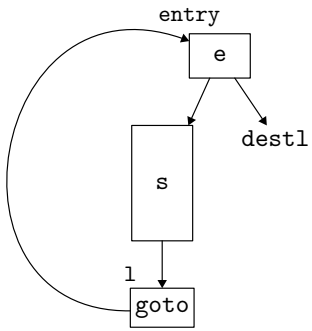
pour une boucle `while`, on crée un cycle dans le graphe de flot de contrôle

```
while (e) {
  ...s...
}
```



Boucle

```
let rec stmt retr s exitl destl = match s with
  | ...
  | Istree.Swhile (e, s) →
      let l = Label.fresh () in
      let entry = condition e (stmt retr s exitl l) destl in
      graph := Label.M.add l (Egoto entry) !graph;
      entry
```



Fonctions

les paramètres d'une fonction et son résultat sont maintenant dans des pseudo-registres

```
type deffun = {
  fun_name : ident;
  fun_formals : register list;
  fun_result : register;
  fun_locals : Register.set;
  fun_entry : label;
  fun_exit : label;
  fun_body : instr Label.map;
}
```

de même pour l'appel

```
type instr =
  ...
  | Ecall of register × ident × register list × label
```


Traduction d'une fonction

la traduction d'une fonction se compose des étapes suivantes

1. on alloue des pseudo-registres frais pour ses arguments, son résultat et ses variables locales
2. on part d'un graphe vide
3. on crée une étiquette fraîche pour la *sortie* de la fonction
4. on traduit le corps de la fonction dans RTL, et le résultat est l'étiquette d'*entrée* de la fonction

Exemple

considérons l'inévitable factorielle

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

on obtient

```
%2 fact(%1)
  entry : L11
  exit  : L1
  locals:
  L11: %8 <- %1   --> L10
  L10: %9 <- 1    --> L9
  L9: ble %8 %9   --> L8, L7
```

```
L8: %2 <- 1                --> L1
L7: %3 <- %1               --> L6
L6: %6 <- %1               --> L5
L5: %7 <- 1                --> L4
L4: %5 <- sub %6 %7        --> L3
L3: %4 <- call fact(%5)    --> L2
L2: %2 <- mul %3 %4        --> L1
```


Phase 3 : ERTL

la troisième phase est la transformation de RTL vers le langage **ERTL** (*Explicit Register Transfer Language*)

objectif : expliciter les **conventions d'appel**, en l'occurrence ici

- les quatre premiers arguments sont passés dans `$a0`, `$a1`, `$a2`, `$a3`, et les suivants sur la pile
- le résultat est renvoyé dans `$v0`
- certains registres sont sauvegardés par l'appelé (`$s0`, `$s1`, ...), les autres par l'appelant (`$v0`, `$a0`, ..., `$t0`, ..., `$ra`)

Registres MIPS

on suppose que le module `Register` décrit aussi les registres physiques

```
type t
...
val parameters : t list  (* pour les n premiers arguments *)
val result : t           (* pour le résultat *)
val ra : t
val callee_saved : t list

(* pour syscall : *)
val a0 : t
val v0 : t
```


Instruction d'appel

dans RTL, on avait

```
  | Ecall of register × ident × register list × label
```

dans ERTL, on a maintenant

```
  | Ecall of ident × int × label
```

i.e. il ne reste que le nom de la fonction à appeler, car de nouvelles instructions vont être insérées pour charger les arguments dans des registres et sur la pile, et pour récupérer le résultat dans `$v0`
(on conserve néanmoins le nombre de paramètres passés dans des registres qui sera utilisé par la phase 4)

de même, les instructions `Emalloc` et `Eprintf` disparaissent au profit d'une nouvelle instruction

```
  | Esyscall of label
```

Nouvelles instructions

les autres instructions de RTL sont inchangées

en revanche, de nouvelles instructions apparaissent :

- pour allouer et désallouer le tableau d'activation

```
| Ealloc_frame of label
| Edelete_frame of label
```

(note : on ne connaît pas encore sa taille)

- pour lire et écrire les paramètres sur la pile

```
| Eget_stack_param of register × int × label
| Eset_stack_param of register × int × label
```

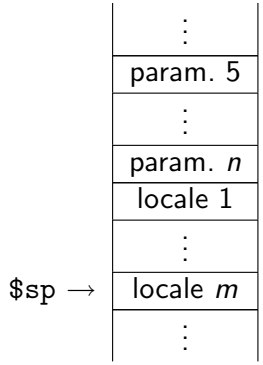
(l'entier est un décalage par rapport au sommet du tableau d'activation)

- le retour est maintenant explicite

```
| Ereturn
```


Tableau d'activation

le tableau d'activation s'organise ainsi :



la zone des m variables locales contiendra tous les pseudo-registres qui ne pourront être stockés dans des registres physiques ; c'est l'allocation de registres (phase 4) qui déterminera m

Insertion des nouvelles instructions

on ne change pas la structure du graphe de flot de contrôle ; on se contente d'insérer de **nouvelles instructions**

- au début de chaque fonction, pour
  - allouer le tableau d'activation
  - sauvegarder `$ra` et les registres *callee-saved*
  - copier les arguments dans les pseudo-registres correspondants
- à la fin de chaque fonction, pour
  - copier le pseudo-registre contenant le résultat dans `$v0`
  - restaurer `$ra` et les registres *callee-saved*
  - désallouer le tableau d'activation
- à chaque appel, pour
  - copier les pseudo-registres contenant les arguments dans `$a0`, ... et sur la pile avant l'appel
  - copier `$v0` dans le pseudo-registre contenant le résultat après l'appel

Traduction

on traduit les instructions de RTL vers ERTL avec une fonction

```
val instr : Rtltree.instr → Ertltree.instr
```

peu de changement, si ce n'est les appels, c'est-à-dire les instructions `Ecall`, `Emalloc` et `Eprintf`

Traduction de l'appel

en RTL, l'appel se présente sous la forme

```
  | Rtltree.Ecall (r, x, rl, l) →
```

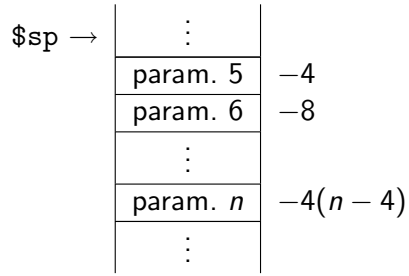
où `r` est le pseudo-registre qui reçoit le résultat, `x` le nom de la fonction et `rl` la liste des pseudo-registres contenant les arguments

on commence par associer les premiers paramètres aux registres physiques de `Register.parameters` :

```
let assoc_formals rl =
  let rec assoc = function
    | [], _ → [], []
    | rl, [] → [], rl
    | r :: rl, p :: pl →
        let a, rl = assoc (rl, pl) in (r, p) :: a, rl
  in
  assoc (formals, Register.parameters)
```


Traduction de l'appel

les paramètres qui ne sont pas associés à des registres physiques sont passés sur la pile ; on les écrit donc à des positions -4, -8, etc. par rapport à `$sp`



on fait donc le choix que l'appelé se chargera seul d'allouer la **totalité** du tableau d'activation (paramètres + locales) par une unique soustraction sur `$sp`

Traduction de l'appel

on se donne

```
let move src dst l = generate (Emunop (dst, Mmove, src, l))
let set_stack r n l = generate (Eset_stack_param (r, n, l))
```

l'appel se traduit alors ainsi (il faut lire le code « à l'envers »)

```
| Rtltree.Ecall (r, x, rl, l) →
    let frl, fsl = assoc_formals rl in
    let n = List.length frl in
    let l = generate (Ecall (x, n, move Register.result r l)) in
    let ofs = ref 0 in
    let l = List.fold_left
      (fun l t → ofs := !ofs - word_size; set_stack t !ofs l)
      l fsl
    in
    let l = List.fold_right (fun (t, r) l → move t r l) frl l in
    Egoto l
```


Primitive `malloc`

pour `malloc`, on utilise l'appel système 9

```
  | Rtltree.Emalloc (r, n, l) →
      Econst (Register.a0, n, generate (
      Econst (Register.v0, 9, generate (
      Esyscall (
      move Register.v0 r l))))
```

Primitive printf

de même pour printf, on utilise les appels systèmes 1 et 11

```
  | Rtltree.Eprintf (r, l) →
      Econst (Register.v0, 1, (
      move r Register.a0 (generate (
      Esyscall (generate (
      Econst (Register.a0, 10, generate (
      Econst (Register.v0, 11, generate (
      Esyscall l))))))))))
```


Traduction des fonctions

il reste à traduire chaque fonction

RTL

```
type deffun = {
  fun_name : ident;
  fun_formals : register list;
  fun_result : register;
  fun_locals : Register.set;
  fun_entry : label;
  fun_exit : label;
  fun_body : instr Label.map;
}
```

ERTL

```
type deffun = {
  fun_name : ident;
  fun_formals : int; (* nb *)

  fun_locals : Register.set;
  fun_entry : label;

  fun_body : instr Label.map;
}
```


Traduction d'une fonction

on associe un pseudo-registre à chaque registre physique qui doit être sauvegardé *i.e.* $ra et les registres de la liste callee_saved

```
let deffun f =
  graph := ...on traduit chaque instruction...
  let savers =
    List.map (fun r → Register.fresh(), r)
    (Register.ra :: Register.callee_saved)
  in
  let entry =
    fun_entry savers f.Rtltree.fun_formals f.Rtltree.fun_entry
  in
  fun_exit savers f.Rtltree.fun_result f.Rtltree.fun_exit;
  { fun_name = f.Rtltree.fun_name;
    ...
    fun_body = !graph; }
```


À l'entrée

à l'entrée de la fonction, il faut

- allouer le tableau d'activation avec Ealloc_frame
- sauvegarder les registres (liste savers)
- copier les arguments dans leurs pseudo-registres (formals)

```
let fun_entry savers formals entry =
  let frl, fsl = assoc_formals formals in
  let ofs = ref 0 in
  let l = List.fold_left
    (fun l t → ofs := !ofs - word_size; get_stack t !ofs l)
    entry fsl
  in
  let l = List.fold_right (fun (t, r) l → move r t l) frl l in
  let l = List.fold_right (fun (t, r) l → move r t l) savers l in
  generate (Ealloc_frame l)
```

(note : l'offset de get_stack est calculé comme pour set_stack pour l'instant)

À la sortie

à la sortie de la fonction, il faut

- copier le pseudo-registre contenant le résultat dans `$v0`
- restaurer les registres sauvegardés
- désallouer le tableau d'activation

```
let fun_exit savers retr exitl =
  let l = generate (Edelete_frame (generate Ereturn)) in
  let l = List.fold_right (fun (t, r) l → move t r l) savers l in
  let l = move retr Register.result l in
  graph := Label.M.add exitl (Egoto l) !graph
```


Exemple : factorielle

```
fact(1)
  entry : L19
  locals: %10, %11, %12
  L19: alloc_frame  --> L18
  L18: %10 <- $ra   --> L17
  L17: %11 <- $s0   --> L16
  L16: %12 <- $s1   --> L15
  L15: %1 <- $a0    --> L11
  L11: %8 <- %1     --> L10
  L10: %9 <- 1      --> L9
  L9: ble %8 %9  --> L8, L7
  L8: %2 <- 1       --> L1
  L7: %3 <- %1      --> L6
  L6: %6 <- %1      --> L5
```

```
L5: %7 <- 1          --> L4
L4: %5 <- sub %6 %7 --> L3
L3: goto L14
L14: $a0 <- %5       --> L13
L13: call fact       --> L12
L12: %4 <- $v0       --> L2
L2: %2 <- mul %3 %4 --> L1
L1: goto L25
L25: $v0 <- %2       --> L24
L24: $ra <- %10      --> L23
L23: $s0 <- %11      --> L22
L22: $s1 <- %12      --> L21
L21: delete_frame    --> L20
L20: return
```

(on suppose que les seuls registres *callee-saved* sont `$s0` et `$s1`)

Déception

c'est encore loin de ce que l'on imagine être un bon code MIPS pour la factorielle

à ce point, il faut comprendre que

- l'allocation de registres (phase 4) tâchera d'associer des registres physiques aux pseudo-registres de manière à limiter l'usage de la pile mais aussi de supprimer certaines instructions ; ainsi, si on réalise `%11` par `$s0`, on supprime tout simplement les deux instructions `L17` et `L23`
- le code n'est pas encore organisé linéairement (le graphe est seulement affiché de manière arbitraire) ; ce sera le travail de la phase 5, qui tâchera notamment de minimiser les sauts

Autre exemple

un autre exemple, avec une fonction ayant plus de 4 arguments

```
int many(int a, int b, int c, int d, int e, int f) {
  if (a == 0) return b; else return many(b, c, d, e, f, a);
}
```

```
many(6)
entry : L31
locals: %17, %18, %19
L31: alloc_frame->L30
L30: %17<-$ra    ->L29
L29: %18<-$s0    ->L28
L28: %19<-$s1    ->L27
L27: %1<-$a0     ->L26
L26: %2<-$a1     ->L25
L25: %3<-$a2     ->L24
L24: %4<-$a3     ->L23
L23: %6<-st(-8) ->L22
L22: %5<-st(-4) ->L13
```

```
L13: %15<-%1 ->L12
L12: %16<-0  ->L11
L11: %14<-seq %15 %16
                ->L10
L10: beqz %14->L8,L9
L8: %8<-%2   ->L7
L7: %9<-%3   ->L6
L6: %10<-%4  ->L5
L5: %11<-%5  ->L4
L4: %12<-%6  ->L3
L3: %13<-%1  ->L2
L2: goto L21
L21: $a0<-%8 ->L20
L20: $a1<-%9 ->L19
```

```
L19: $a2<-%10   ->L18
L18: $a3<-%11   ->L17
L17:st(-8)<-%13 ->L16
L16:st(-4)<-%12 ->L15
L15: call many  ->L14
L14: %7<-$v0    ->L1
L1: goto L37
L37: $v0<-%7    ->L36
L36: $ra<-%17   ->L35
L35: $s0<-%18   ->L34
L34: $s1<-%19   ->L33
L33:delete_frame->L32
L32: return
L9: %7<-%2      ->L1
```

Appel terminal

c'est au niveau de la traduction RTL → ERTL qu'il faut réaliser l'optimisation des **appels terminaux** si on le souhaite

en effet, les instructions à produire ne sont pas les mêmes, et ce changement aura une influence dans la phase suivante d'allocation des registres

Appel terminal

il y a une difficulté cependant, si la fonction appelée par un appel terminal n'a pas le même nombre d'arguments passés sur la pile ou de variables locales, car la tableau d'activation doit être modifié

deux solutions au moins

- limiter l'optimisation de l'appel terminal aux cas où le tableau d'activation n'est pas modifié : c'est le cas s'il s'agit d'un appel terminal d'une fonction récursive à elle-même
- l'appelant modifie le tableau d'activation et transfère le contrôle à l'appelé *après* l'instruction de création de son tableau d'activation

Phase 4 : LTL

la quatrième phase est la traduction de ERTL vers **LTL** (*Location Transfer Language*

il s'agit de faire disparaître les pseudo-registres au profit

- de registres physiques, de préférence
- d'emplacements de pile, sinon

c'est ce que l'on appelle l'**allocation de registres**

Allocation de registres

l'allocation de registres est une phase complexe, que l'on va elle-même décomposer en plusieurs étapes

1. analyse de **durée de vie**
   - il s'agit de déterminer à quels moments précis la valeur d'un pseudo-registre est nécessaire pour la suite du calcul
2. construction d'un **graphe d'interférence**
   - il s'agit d'un graphe indiquant quels sont les pseudo-registres qui ne peuvent pas être réalisés par le même emplacement
3. allocation de registres par **coloration de graphe**
   - c'est l'affectation proprement dite de registres physiques et d'emplacements de pile aux pseudo-registres

Analyse de durée de vie

dans la suite on appelle *variable* un pseudo-registre ou un registre physique

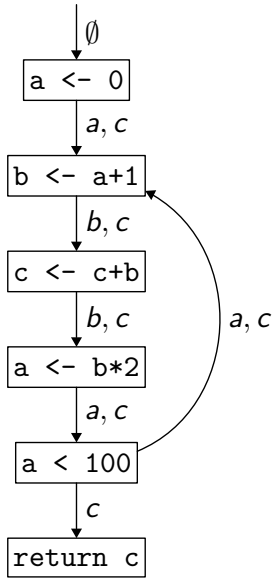
Définition (variable vivante)

En un point de programme, une variable est dite **vivante** *si la valeur qu'elle contient peut être utilisée dans la suite de l'exécution.*

c'est donc une notion naturellement attachée aux *arêtes* du graphe de flot de contrôle que nous avons construit

Exemple

```
    a <- 0
L1: b <- a + 1
    c <- c + b
    a <- b * 2
    if a < 100 goto L1
    return c
```



Définitions et utilisations

la notion de variable vivante se déduit des **définitions** et des **utilisations** des variables effectuées par chaque instruction

Définition

Pour une instruction l du graphe de flot de contrôle, on note

- *$def(l)$ l'ensemble des variables définies par cette instruction, et*
- *$use(l)$ l'ensemble des variables utilisées par cette instruction.*

exemple : pour l'instruction $l \equiv r_1 \leftarrow \text{add } r_2\ r_3$ on a

$$def(l) = \{r_1\} \quad \text{et} \quad use(l) = \{r_2, r_3\}$$

Calcul des variables vivantes

pour calculer les variables vivantes, il est commode de les associer non pas aux arêtes mais plutôt aux *nœuds* du graphe de flot de contrôle, c'est-à-dire à chaque instruction

mais il faut alors distinguer les variables **vivantes à l'entrée** d'une instruction et les variables **vivantes à la sortie**

Définition

Pour une instruction l du graphe de flot de contrôle, on note

- *$in(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes arrivant sur l, et*
- *$out(l)$ l'ensemble des variables vivantes sur l'ensemble des arêtes sortant de l.*

Équations

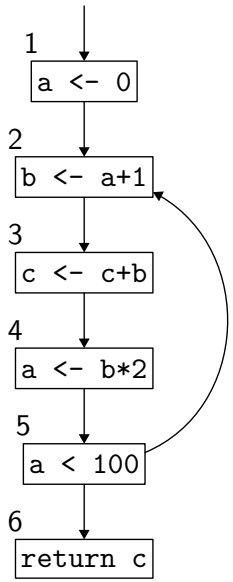
les équations qui définissent $in(l)$ et $out(l)$ sont les suivantes

$$\begin{cases} in(l) & = & use(l) \cup (out(l) \backslash def(l)) \\ out(l) & = & \bigcup_{s \in succ(l)} in(s) \end{cases}$$

il s'agit d'équations récursives dont la plus petite solution est celle qui nous intéresse

nous sommes dans le cas d'une fonction monotone sur un domaine fini et nous pouvons donc appliquer le théorème de Tarski

Calcul du point fixe



$$\begin{cases} in(l) & = & use(l) \cup (out(l) \backslash def(l)) \\ out(l) & = & \bigcup_{s \in succ(l)} in(s) \end{cases}$$

	use	def	in	out	in	out		in	out
1		a				a	...	c	a,c
2	a	b	a		a	b,c	...	a,c	b,c
3	b,c	c	b,c		b,c	b	...	b,c	b,c
4	b	a	b		b	a	...	b,c	a,c
5	a		a	a	a	a,c	...	a,c	a,c
6	c		c		c		...	c	

on obtient le point fixe après 7 itérations

Calcul du point fixe

en supposant un graphe de flot de contrôle contenant N sommets et N variables, un calcul brutal a une complexité $O(N^4)$ dans le pire des cas

on peut améliorer l'efficacité du calcul de plusieurs façons

- en calculant dans l'« ordre inverse » du graphe de flot de contrôle, et en calculant *out* avant *in* (sur l'exemple précédent, on converge alors en 3 itérations au lieu de 7)
- en fusionnant les sommets qui n'ont qu'un unique prédécesseur et qu'un unique successeur avec ces derniers (*basic blocks*)
- en utilisant un algorithme plus subtil qui ne recalcule que les valeurs de *in* et *out* qui peuvent avoir changé ; c'est l'algorithme de Kildall

Algorithme de Kildall

idée : si $in(l)$ change, alors il faut refaire le calcul pour les prédécesseurs de l uniquement

$$\begin{cases} out(l) & = & \bigcup_{s \in succ(l)} in(s) \\ in(l) & = & use(l) \cup (out(l) \backslash def(l)) \end{cases}$$

d'où l'algorithme suivant

```
soit WS un ensemble contenant tous les sommets
tant que WS n'est pas vide
    extraire un sommet l de WS
    old_in ← in(l)
    out(l) ← ...
    in(l)  ← ...
    si in(l) est différent de old_in(l) alors
        ajouter tous les prédécesseurs de l dans WS
```

Calcul de *def* et *use*

le calcul des ensemble $def(l)$ (définitions) et $use(l)$ (utilisations) est immédiat pour la plupart des instructions

exemples

```
let def_use = function
  | Econst (r,_,_)          → [r],  []
  | Eassign_global (r,_,_) → [],   [r]
  | Emunop (rd,_,rs,_)      → [rd], [rs]
  | Egoto _                 → [],   []
  | ...
```


Calcul de *def* et *use*

il est un peu plus subtil en ce qui concerne les appels

pour un appel, on exprime que les $min(4, n)$ premiers registres de la liste `parameters` vont être utilisés, et que tous les registres *caller-saved* peuvent être écrasés par l'appel

```
  | Ecall (_,n,_) →
      caller_saved, prefix n parameters
```

pour un appel système, seuls `$a0` et `$v0` sont affectés

```
  | Esyscall l →
      [v0], [a0; v0]
```

enfin pour `return`, `$v0`, `$ra` et tous les registres *callee-saved* vont être utilisés

```
  | Ereturn →
      [], result :: ra :: callee_saved
```


Exemple

reconsidérons la forme ERTL de la factorielle

```
fact(1)
  entry : L19
  locals: %10, %11, %12
  L19: alloc_frame  --> L18
  L18: %10 <- $ra   --> L17
  L17: %11 <- $s0   --> L16
  L16: %12 <- $s1   --> L15
  L15: %1 <- $a0    --> L11
  L11: %8 <- %1     --> L10
  L10: %9 <- 1      --> L9
  L9: ble %8 %9  --> L8, L7
  L8: %2 <- 1       --> L1
  L7: %3 <- %1      --> L6
  L6: %6 <- %1      --> L5
```

```
L5: %7 <- 1           --> L4
L4: %5 <- sub %6 %7   --> L3
L3: goto L14
L14: $a0 <- %5        --> L13
L13: call fact        --> L12
L12: %4 <- $v0        --> L2
L2: %2 <- mul %3 %4   --> L1
L1: goto L25
L25: $v0 <- %2        --> L24
L24: $ra <- %10       --> L23
L23: $s0 <- %11       --> L22
L22: $s1 <- %12       --> L21
L21: delete_frame     --> L20
L20: return
```


L'exemple de la factorielle

l'analyse de durée de vie donne

```
L19: alloc_frame  --> L18     in=$a0,$ra,$s0,$s1       out=$a0,$ra,$s0,$s1
L18: %10 <- $ra   --> L17     in=$a0,$ra,$s0,$s1       out=$a0,$s0,$s1
L17: %11 <- $s0   --> L16     in=$a0,$s0,$s1,%10       out=$a0,$s1,%10,%11
L16: %12 <- $s1   --> L15     in=$a0,$s1,%10,%11       out=$a0,%10,%11,%12
L15: %1 <- $a0    --> L11     in=$a0,%10,%11,%12       out=%1,%10,%11,%12
L11: %8 <- %1     --> L10     in=%1,%10,%11,%12        out=%1,%10,%11,%12,%8
L10: %9 <- 1      --> L9      in=%1,%10,%11,%12,%8     out=%1,%10,%11,%12,%8,%9
L9: ble %8 %9     --> L8, L7  in=%1,%10,%11,%12,%8,%9  out=%1,%10,%11,%12
L8: %2 <- 1       --> L1      in=%10,%11,%12           out=%10,%11,%12,%2
L1: goto L25                  in=%10,%11,%12,%2        out=%10,%11,%12,%2
L25: $v0 <- %2    --> L24     in=%10,%11,%12,%2        out=$v0,%10,%11,%12
L24: $ra <- %10   --> L23     in=$v0,%10,%11,%12       out=$ra,$v0,%11,%12
L23: $s0 <- %11   --> L22     in=$ra,$v0,%11,%12       out=$ra,$s0,$v0,%12
L22: $s1 <- %12   --> L21     in=$ra,$s0,$v0,%12       out=$ra,$s0,$s1,$v0
L21: delete_frame --> L20     in=$ra,$s0,$s1,$v0       out=$ra,$s0,$s1,$v0
L20: return                   in=$ra,$s0,$s1,$v0       out=
L7: %3 <- %1      --> L6      in=%1,%10,%11,%12        out=%1,%10,%11,%12,%3
L6: %6 <- %1      --> L5      in=%1,%10,%11,%12,%3     out=%10,%11,%12,%3,%6
L5: %7 <- 1       --> L4      in=%10,%11,%12,%3,%6     out=%10,%11,%12,%3,%6,%7
L4: %5 <- sub %6 %7-->L3      in=%10,%11,%12,%3,%6,%7  out=%10,%11,%12,%3,%5
L3: goto L14                  in=%10,%11,%12,%3,%5     out=%10,%11,%12,%3,%5
L14: $a0 <- %5    --> L13     in=%10,%11,%12,%3,%5     out=$a0,%10,%11,%12,%3
L13: call fact    --> L12     in=$a0,%10,%11,%12,%3    out=$v0,%10,%11,%12,%3
L12: %4 <- $v0    --> L2      in=$v0,%10,%11,%12,%3    out=%10,%11,%12,%3,%4
L2: %2 <- mul %3 %4-->L1      in=%10,%11,%12,%3,%4     out=%10,%11,%12,%2
```

La semaine prochaine

la semaine prochaine

- TD 10 le mercredi 16 décembre
  - coloriage de graphe
- Cours 11 le jeudi 17 décembre
  - production de code efficace, partie 2

Remerciements

ces notes s'inspirent largement d'un cours de François Pottier (cours INF553 à l'École Polytechnique) ; l'architecture est celle du compilateur CompCert de Xavier Leroy