Optimisation I [1]
23 novembre 2011

1. Production de code MIPS optimisé
   - Cas d'étude: un sous-ensemble de C
   - Phase 1 : Production d'un ast
   - Phase 2: RTL
   - Phase 3: ERTL
2. Analyse de flots de données
   - Exemples d'analyse de flots
   - Théorèmes de point fixe
   - Algorithmes

1. Ce cours s'inspire des chapitres 10 et 11 du cours de compilation de J.-C. Filliâtre (ENS)

Critères pour un bon code

- Rapidité d'exécution
- Sécurité
- Taille du code
- Consommation d'énergie

Remarques:

- Ces critères peuvent être contradictoires entre eux.
- Trouver un programme compilé optimal est un problème indécidable.
- On se contente d'heuristiques.
- Des méthodes dynamiques d'optimisation permettent de choisir les options de compilation en fonction du comportement du programme sur des données bien choisies.

Dans ce cours

Produire du code MIPS qui utilise intelligemment les registres.

Allocation de registres I

Une exploitation efficace des registres nécessite d'introduire des langages intermédiaires

1. Sélection d'instructions sur la forme arborescente
2. RTL (Register Transfer Language)
   - Infinité de registres
   - Instructions élémentaires entre registres
   - Graphe de flot de contrôle
3. ERTL (Explicit Register Transfer Language)
   Comme RTL mais en explicitant le passage des arguments et du résultat dans les appels

Ces deux premières phases introduisent de nombreux registres et instructions (**move**, **goto**) inutiles

Allocation de registres II

4. LTL (Location Transfer Language)
   - allocation de registres physiques ou mise dans la pile
   - utilise une analyse de durée de vie des variables
   - construit un graphe d'interférences et de préférences qui est colorié
5. Linéarisation
   - Reconstruire un code linéaire en minimisant les sauts.

Ces deux dernières phases reconstruisent un code efficace.

Sommaire

Langage mini-C

- On considère un sous-ensemble de C
  - entiers
  - pointeurs sur des structures
  - commande `print` pour les entiers

Exemple

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

struct L { int val; struct L * next; };
int printL (struct L * l){
   while (l!=0) {print(l->val); l=l->next;}
   return 0;
}
```

- Tous les objets ont une taille unitaire

Syntaxe concrète de mini-C

$$
\begin{array}{rcl}
E & ::= & n \\
 & | & L \\
 & | & L = E \\
 & | & E\ op\ E \mid - E \mid \texttt{!}\ E \\
 & | & x\texttt{(}E,\ldots,E\texttt{)} \\
 & | & \texttt{malloc(sizeof(struct}\ x\texttt{))} \\
L & ::= & x \\
 & | & E\texttt{->}x \\
op & \rightarrow & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
 & | & \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
D & ::= & V \\
 & | & T\ x(T\ x,\ldots,T\ x)\ B \\
 & | & \texttt{struct}\ x\ \texttt{\{}\ V \ldots V\ \texttt{\}}\texttt{;}
\end{array}
$$

$$
\begin{array}{rcl}
S & ::= & E\texttt{;} \\
 & | & \texttt{if (}E\texttt{)}\ S \\
 & | & \texttt{if (}E\texttt{)}\ S\ \texttt{else}\ S \\
 & | & \texttt{while (}E\texttt{)}\ S \\
 & | & \texttt{return}\ E\texttt{;} \\
 & | & \texttt{print(}E\texttt{);} \\
 & | & B \\
B & ::= & \texttt{\{}\ V \ldots V\ S \ldots S\ \texttt{\}} \\
V & ::= & T\ x,\ldots,x\texttt{;} \\
T & ::= & \texttt{int} \mid \texttt{struct}\ x\ \texttt{*} \\
P & ::= & D \ldots D
\end{array}
$$

Arbres de syntaxe abstraite

Les types sont soit des entiers soit des structures:

```
type typ = TInt | Struct of string
```

Le langage C distingue expressions et instructions.
Une valeur gauche est une variable locale ou globale ou bien le champs d'une expression

```
type expr =
    | Cst of int
    | Access of left
    | Assign of left * expr
    | Binop of binop * expr * expr
    | Unop of unop * expr
    | Call of string * expr list
    | Malloc of int
and left = Loc of string | Glob of string
          | Field of expr * int
```

Arbres de syntaxe abstraite (2)

```
type instr =
  | Skip | Expr of expr
  | If of expr * instr * instr
  | While of expr * instr
  | Block of (decl_var list * instr list)
  | Return of expr
  | Print of expr
and decl_var = string * typ
```

Une fonction a un nom, des paramètres, des variables locales et une suite d'instructions.

```
type decl_fun =
  {name:string; params : decl_var list; return:typ;
   locals: decl_var list; body: instr list}
```

Un programme contient des variables globales et des fonctions.

```
type program = {globs:decl_var list; funs:decl_fun list}
```

Sommaire

Production de l'ast

- Analyse lexicale et syntaxique
- Analyse de portée/typage
  - création de noms uniques
  - distinction variables globales et locales
  - affectation d'un décalage à chaque champs d'une structure
- Phase de transformation de l'arbre pour la sélection d'instructions

Sélection d'instructions

- L'assembleur MIPS propose des instructions arithmétiques optimisées pour des cas particuliers:
  - **addi** pour ajouter à un registre une constante de taille 16 bits
  - **slti** pour comparer un registre à une constante de taille 16 bits
- On peut également effectuer des précalculs à la compilation.

$$\textbf{not}(e_1 < e_2) = e_1 \geq e_2$$

Attention

- Certaines transformations arithmétiques ne sont correctes qu'en l'absence d'effets de bord.

$$0 \times e = 0$$

- Les calculs sur les flottants ne satisfont pas non plus les règles usuelles (associativité . . .)

Transformations

- Transformer l'arbre de syntaxe abstraite :
  - On ajoute des opérateurs supplémentaires pour les opérations spécialisées.
  - Exemple: `Addi of int` comme nouvelle opérateur unaire.
- Les transformations s'expriment comme des règles de réécriture:
  e est une expression quelconque et n une constante immédiate.

$$
\begin{array}{rcll}
e + 0 & \longrightarrow & e & \\
0 + e & \longrightarrow & e & \\
e - 0 & \longrightarrow & e & \\
e + n & \longrightarrow & \textbf{Addi}(n)\ e & \\
e - n & \longrightarrow & \textbf{Addi}(-n)\ e & \\
n + e & \longrightarrow & \textbf{Addi}(n)\ e & \\
n_1 \ \mathrm{op}\ n_2 & \longrightarrow & n & n \text{ résultat du calcul}
\end{array}
$$

Règles de réecriture

Propriétés attendues

- Préservation de la sémantique:
  Attention aux effets de bord si l'ordre d'exécution des expressions est modifié.
- Terminaison (trouver un ordre qui décroit).
- Confluence (une seule forme simplifiée, l'ordre des simplifications ne joue pas)

On cherche à faire remonter les additions constantes en tête des expressions:

$$\begin{aligned}(e_1 + n_1) + (e_2 + n_2) &\longrightarrow (e_1 + e_2) + (n_1 + n_2)\\ (e_1 + n) + e_2 &\longrightarrow (e_1 + e_2) + n\\ e_1 + (e_2 + n) &\longrightarrow (e_1 + e_2) + n\end{aligned}$$

Implantation

- On cherche à construire des expressions qui ne peuvent plus être simplifiées.
- On introduit des fonctions de construction comme *mkadd* qui effectuent les simplifications.
- Les arguments eux-mêmes sont supposés en forme simplifiée.

```
let rec translate = function
 Binop(Add,e1,e2) -> let e1' = translate e1
                     and e2' = translate e2 in
                     mkadd e1' e2'
| ...

let int16 n = -32768 <= n && n < 32767
```

Implantation constructeur

```
let mkadd e1 e2 =
    match e1,e2 with
    Cte(0),e | e,Cte(0) -> e
  | Cte(n1),Cte(n2) -> Cte(n1+n2)
  | Cte(n1),Unop(Addi(n2),e) | Unop(Addi(n2),e),Cte(n1) |
            -> mkadd (Cte(n1+n2)) e
  | Cte(n),e | e,Cte(n) when int16 n
            -> Unop(Addi(n),e)
  | Unop(Addi(n1),e1),Unop(Addi(n2),e2)  when (n1+n2=0)
            -> mkadd e1 e2
  | Unop(Addi(n1),e1),Unop(Addi(n2),e2)  when (int16 n)
            -> Unop(Addi(n1+n2),mkadd e1 e2)
  | Unop(Addi(n),e1),e2 | e1,Unop(Addi(n),e2)
            -> Unop(Addi(n),mkadd e1 e2)
  | _       -> Binop(Add,e1,e2)
```

Sommaire

RTL:Register Transfer Language

- Introduction d'instructions portant sur les registres
- Une infinité de (pseudo) registres
- Les expressions disparaissent et deviennent des instructions $r \leftarrow e$
- Création d'un graphe de flot de contrôle:
  - Chaque instruction contient l'étiquette de l'instruction suivante (deux étiquettes pour les branchements conditionnels)
  - Le graphe associe une instruction à chaque étiquette.
- Les appels de fonctions sont vus de manière macroscopique: on se contente de mettre les arguments et la valeur de retour dans des registres.
- Une définition de fonction utilise des registres frais pour ses arguments et le résultat.
  Elle est associée à un graphe dont on distingue les étiquettes d'entrée et de sortie.

RTL:Gestion des variables

- les variables globales restent des noms.
  Elles seront allouées dans la zone de données de l'assembleur et repérées par une étiquette.
- Les variables locales sont associées à des (pseudo)registres uniques fixés.
  Elles seront ensuite stockées dans des registres physiques ou dans la pile.
- Des pseudo-registres servent pour les calculs intermédiaires.
- Les structures sont allouées dans le tas, les accès et mises à jour utilisent les commandes **lw** et **sw**.

Registres

```
type reg  (* type abstrait des registres *)
   val fresh : unit -> reg  (* pseudo registre *)
(* pseudo registre pour une variable locale *)
   val var : string -> reg
(* registres spécifiques *)
   val tmp1 : reg     (* accès à la pile *)
   val tmp2 : reg     (* accès à la pile *)
   val sp : reg
   val ra : reg
   val v0 : reg
   val a0 : reg
(* registres de la machine *)
   val is_hw : reg -> bool
(* registres réservés pour les arguments *)
   val params : reg list
(* registres temporaires utilisables *)
   val caller_saved : reg list
   val callee_saved : reg list
```

Registres

Données structurées utilisant des registres:

```
(* ensemble de registres *)
module S : Set.S with type elt = reg

(* map indexées par des registres *)
module M : Map.S with type key = reg
```

RTL: arbres de syntaxe abstraites

```
type instr =
  | Econst   of reg * int * label
  | Eunop    of reg * unop * reg * label
  | Ebinop   of reg * binop * reg * reg * label
  | EGaccess of reg * string * label
  | EGassign of reg * string * label
  | Eload    of reg * reg * int * label
  | Estore   of reg * reg * int * label
  | Emove    of reg * reg * label
  | Emalloc  of reg * int * label
  | Eprint   of reg * label
  | Eubranch of ubranch * reg * label * label
  | Ebbranch of bbranch * reg * reg * label * label
  | Ecall    of reg * string * reg list * label
  | Egoto    of label
```

Construction du graphe

- Le graphe est représenté comme une map des étiquettes dans les instructions.
- Le graphe est construit en ajoutant les instructions en général associées à une nouvelle étiquette.

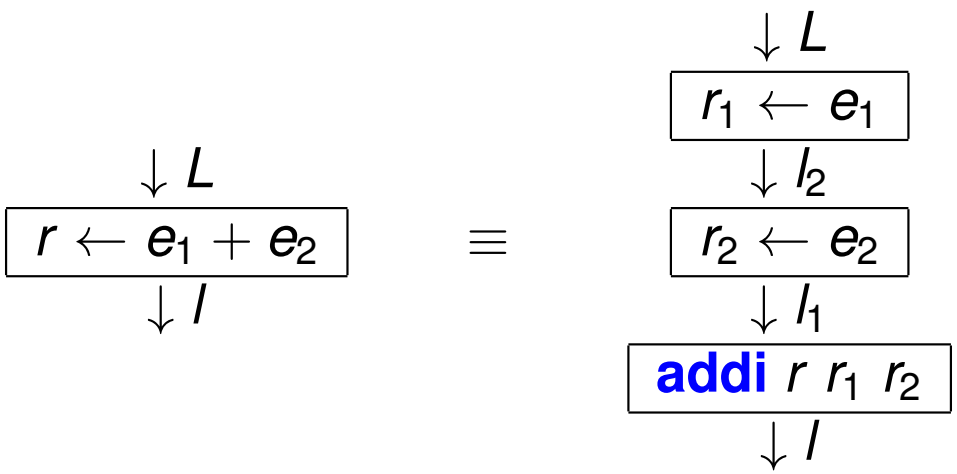
```
type graph = instr Label.M.t

let graph:graph ref = ref Label.M.empty
(* remise à zéro du graphe *)
let clear_graph () = graph := Label.M.empty
(* ajout d'une arête *)
let add_lab_instr lab i =
  graph := Label.M.add lab i !graph
(* création d'une étiquette et ajout de l'instruction *)
let add_instr i =
  let lab = Label.fresh () in add_lab_instr lab i; lab
```

Traduction des expressions

- La fonction de traduction `rtl_expr` prend en arguments:
  - l'expression e à calculer (de type `expr`)
  - le registre r (de type `reg`) dans lequel on calcule l'expression.
  - l'étiquette l (de type `label`) de la suite du calcul
- Elle renvoie une nouvelle étiquette pour démarrer le calcul
- on construit donc les instructions à l'envers

Exemple r_1 et r_2 sont de nouveaux registres:



Code pour la traduction d'expressions

Expressions simples

```
let rec rtl_expr reg e l = match e with
  | Cst(n) -> add_instr (Econst(reg,n,l))
  | Unop(op,e) ->
      let reg1 = Register.fresh() in
      let l1 = add_instr (Eunop(reg,op,reg1,l)) in
      rtl_expr reg1 e l1
  | Binop(op,e1,e2) ->
      let reg1 = Register.fresh()
      and reg2 = Register.fresh() in
      let l2 = add_instr (Ebinop(reg,op,reg1,reg2,l)) in
      let l1 = rtl_expr reg2 e2 l2 in
      rtl_expr reg1 e1 l1
   | Malloc(n) -> add_instr (Emalloc(reg,n,l))
...
```

Code pour la traduction d'expressions (2)

Accès aux variables locales et globales

```
...
  | Access(Glob s) -> add_instr (EGaccess(reg,s,l))
  | Assign(Glob s,e) ->
      let l1 = add_instr (EGassign(reg,s,l))
      in rtl_expr reg e l1
  | Access(Loc s) ->
       let regs = Register.var s in
       add_instr (Emove(reg,regs,l))
  | Assign(Loc s,e) ->
      let regs = Register.var s in
      let l1 = add_instr (Emove(regs,reg,l))
      in rtl_expr reg e l1
...
```

Code pour la traduction d'expressions (3)

Accès aux champs de structure et appel de fonctions

```
...
| Access(Field(e,i)) -> let reg1 = Register.fresh() in
   let l1 = add_instr (Eload(reg,reg1,wsize * i,l))
   in rtl_expr reg1 e l1
| Assign(Field(e1,f),e2) ->
   let reg1 = Register.fresh() in
   let l1 = add_instr (Estore(reg,reg1,wsize*f,l)) in
   let l2 = rtl_expr reg e2 l1 in
   rtl_expr reg1 e1 l2
| Call(f,le) ->
   let lr = List.map (fun _ -> Register.fresh ()) le in
   let l0 = add_instr (Ecall(reg,f,lr,l)) in
   List.fold_right2 (fun e r l -> rtl_expr r e l) le lr l0
```

Traduction des instructions

- Les instructions forment le corps des fonctions
- On doit prendre en compte la commande **return** qui calcule un résultat et sort de la fonction.
- La fonction `rtl_instr` de traduction des instructions:
  - prend en argument le registre pour la valeur de retour (`rreturn`) ainsi que l'étiquette de sortie (`lexit`) en plus de l'étiquette de l'instruction suivante `lab`).
  - retourne l'étiquette pour débuter le calcul.

Code pour les instructions

Instructions élémentaires

```
let rec rtl_instr rreturn i lexit lab = match i with
    Skip -> lab
  | Return e -> rtl_expr rreturn e lexit
  | Expr e -> let reg = Register.fresh() in
     rtl_expr reg e lab
  | Print e ->
     let reg = Register.fresh() in
     let lab1 = add_instr (Eprint(reg,lab)) in
     rtl_expr reg e lab1
  | Block ( _ , li ) ->
     List.fold_right
      (fun i l -> rtl_instr rreturn i lexit l)
      li lab
...
```

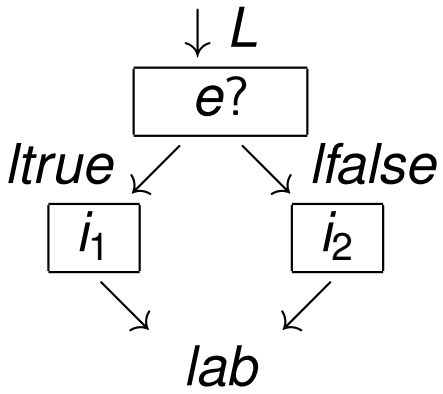
Instructions conditionnelles

- On optimise le calcul des conditions en manipulant les étiquettes.
- La valeur de la condition n'est pas rendue dans un registre mais par le choix entre deux étiquettes.
- La fonction de traduction prend en argument l'étiquette de branchement si la condition est vraie et celle si la condition est fausse.
- Les comparaisons arithmétiques sont remplacées par les instructions de branchement binaires **blt**, **ble**.
- Le cas général est traité par un test à zéro (cas **false**) de la condition.

Code pour les conditions

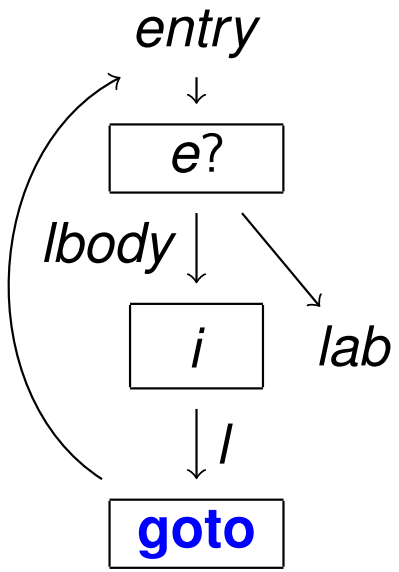
```
let rec rtl_cond e ltrue lfalse = match e with
| Binop(And,e1,e2) ->
    let l = rtl_cond e2 ltrue lfalse in
    rtl_cond e1 l lfalse
| Binop(Or,e1,e2) ->
    let l = rtl_cond e2 ltrue lfalse in
    rtl_cond e1 ltrue l
| Unop(Neg,e) -> rtl_cond e lfalse ltrue
| Binop(op,e1,e2) when is_rel op ->
    let bop = branch_rel op in
    let r1 = Register.fresh() and r2 = Register.fresh() in
    let l1 = add_instr (Ebbranch(bop,r1,r2,ltrue,lfalse)) in
    let l2 = rtl_expr r2 e2 l1 in
    rtl_expr r1 e1 l2
| e -> let r = Register.fresh() in
    let l = add_instr (Eubranch(Beqz,r,lfalse,ltrue)) in
    rtl_expr r e l
```

Compilation des conditionnelles




```
let rec rtl_instr rreturn i lexit lab = match i with
...
  | If(e,i1,i2) ->
      let ltrue = rtl_instr rreturn i1 lexit lab
      and lfalse =  rtl_instr rreturn i2 lexit lab
      in rtl_cond e ltrue lfalse
...
```

Compilation des boucles




```
let rec rtl_instr rreturn i lexit lab = match i with
...
| While(e,i) -> let l = Label.fresh () in
    let lbody = rtl_instr rreturn i lexit l in
    let entry = rtl_cond e lbody lab in
      add_lab_instr l (Egoto entry);
      entry
```

Transformation des fonctions

- On oublie les types
- On garde les registres utilisés pour les paramètres et le retour
- On garde les étiquettes d'entrée et de sortie du code

```
type decl_fun =
  { name : string;
    params: register list;
    return:register;
    entry:label; exit:label;
    body : graph
   }
```

Transformation des déclarations

```
let rtl_fun df =
  let rreturn = Register.fresh() in
  let lexit = Label.fresh() in
  clear_graph();
  let lentry = List.fold_right
       (fun i l -> rtl_instr rreturn i lexit l)
        df.Ast.body lexit
  in { name=df.Ast.name;
       params=List.map (fun (s,_) -> Register.var s)
                df.Ast.params;
       return=rreturn;
       entry=lentry;
       exit=lexit;
       body= !graph
     }
```

Sommaire

ERTL:Extended Register Transfer Language

- Gestion atomique de l'appel de fonctions:
  - Passage des paramètres dans les registres dédiés ($a0,$a1,$a2,$a3) ou dans la pile
  - Récupération du résultat dans $v0
- On garde trace du nombre d'arguments passés dans des registres dédiés (dans l'appel et la déclaration de fonction)
- Nouvelles instructions:
  - Instructions pour l'allocation et la desallocation du tableau d'activation sur la pile
  - Instructions pour l'accès et la mise à jour dans la pile
  - Instruction pour le retour de fonction **return**
  - Compilation des instructions **print** et **malloc** en appel **syscall**

Arbre de syntaxe abstraite

Instruction analogues

```
type instr =
   | Econst of reg * int * label
   | Eunop of reg * unop * reg * label
   | Ebinop of reg * binop * reg * reg * label
   | EGassign of reg * string * label
   | EGaccess of reg * string * label
   | Eload of reg * reg * int * label
   | Estore of reg * reg * int * label
   | Emove of reg * reg * label
   | Eubranch of ubranch * reg * label * label
   | Ebbranch of bbranch * reg * reg * label * label
   | Egoto of label
...
```

Arbre de syntaxe abstraite

Nouvelles Instructions

```
type instr =
...
  | Ecall of string * int * label
  | Ealloc_frame of label
  | Edel_frame of label
  | Ereturn
  | Eget_param of reg * int * label
  | Eset_param of reg * int * label
  | Esyscall of label
```

Transformation de **malloc**

malloc(r,n) doit être transformé en:

```
$a0 ← n
$v0 ← 9
syscall
r ← $v0
```

```
let ertl_instr = function
...
    | Rtl.Emalloc(r,n,l) ->
        let lab1 =  add_instr (Emove(r,v0,l)) in
        let lab2 =  add_instr (Esyscall(lab1)) in
        let lab3 =  add_instr (Econst(v0,9,lab2)) in
            (Econst(a0,n,lab3))
```

Transformation de **print**

print(r) : impression de la valeur puis d'un retour à la ligne:

```
$a0 ← r
$v0 ← 1
syscall
$a0 ← 10
$v0 ← 11
syscall
```

```
let ertl_instr = function
...
| Rtl.Eprint(r,l)->
      let lab1 = add_instr (Esyscall l) in
      let lab2 = add_instr (Econst(v0,11,lab1)) in
      let lab3 = add_instr (Econst(a0,10,lab2)) in
      let lab4 = add_instr (Esyscall lab3) in
      let lab5 = add_instr (Econst(v0,1,lab4)) in
       (Emove(a0,r,lab5))
```

Appel de fonction

$r \leftarrow$ **call**(f, lr)

- On place n ($n \leq 4$) paramètres dans les registres ($\$a0$,$\$a1$,...)
  $\$a0 \leftarrow$r1
  $\$a1 \leftarrow$r2
  ...
- On place si nécessaire les autres paramètres sur la pile
  sw r5 -4($\$sp$)
  sw r6 -8($\$sp$)
  ...
  L'allocation du tableau d'activation sera faite en une fois par l'appelé pour les paramètres + les données locales
- On fait l'appel à la fonction **call**(f, n)
- On affecte à $\$r$ le résultat qui est calculé dans $\$v0$: $r \leftarrow \$v0$

Code de l'appel de fonction

```
let ertl_instr = function
  | Rtl.Ecall(r,f,rl,l)  ->
      let ass,rls = assoc_formals rl in
      let n = List.length ass in
      let lab1 = add_instr (Emove(r,v0,l)) in
      let lab2 =  add_instr (Ecall(f,n,lab1)) in
      let ofs = ref 0 in
      let labp = List.fold_left
          (fun lab ri -> ofs := !ofs - wsize;
             add_instr (Eset_param(ri,!ofs,lab)))
          lab2 rls
      in
      let laba = List.fold_right
          (fun (ri,rd) lab -> add_instr (Emove(rd,ri,lab)))
          ass labp
      in  (Egoto(laba))
```

Traduction du graphe de flot

- Les autres instructions sont traduites à l'identique

```
let ertl_instr = function
  | Rtl.Econst(r,n,l)            -> Econst(r,n,l)
  | Rtl.Eunop(r1,op,r2,l)        -> Eunop(r1,op,r2,l)
  | Rtl.Ebinop(r1,op,r2,r3,l)    -> Ebinop(r1,op,r2,r3,l)
  | Rtl.EGassign(r,x,l)          -> EGassign(r,x,l)
...

let ertl_graph g =
  Label.M.map
    (fun lab i -> add_lab_instr lab (ertl_instr i))
    g
```

Définition de fonctions

À l'entrée de la fonction

- On met une instruction d'allocation de frame
- On met à jour les pseudos-registres paramètres de la fonction (à partir des registres $\$a_i$ ou de la pile)
- On sauvegarde les registres à sauvegarder par l'appelé

Accès dans la pile

- La taille de pile nécessaire pour l'exécution de la fonction sera connue après l'étape LTL
- On peut repérer les objets à partir du sommet de pile (pas besoin du registre $\$fp$)

À la sortie de la fonction

- On met la valeur du registre de retour de la fonction dans $\$v0$
- On restaure les registres sauvegardés
- On détruit la frame allouée
- On exécute l'instruction **return**

Traitement de l'entrée dans la fonction

```
let fun_entry savers formals entry =
let ass,rls = assoc_formals formals in
let ofs = ref 0 in
let labp = List.fold_left
        (fun lab ri -> ofs := !ofs - wsize;
           add_instr (Eget_param(ri,!ofs,lab)))
        entry rls
in
let laba = List.fold_right
        (fun (ri,rd) l -> add_instr (Emove(ri,rd,l)))
        ass labp
in
let labs = List.fold_right
        (fun (ri,rd) l -> add_instr (Emove(ri,rd,l)))
        savers laba
in add_instr (Ealloc_frame labs)
```

Traitement de la sortie de fonction

```
let fun_exit savers retr lexit =
  let lab1 = add_instr (Edel_frame (add_instr Ereturn)) in
  let labs = List.fold_right
          (fun (ri,rd) lab -> add_instr (Emove(rd,ri,lab)))
          savers lab1
  in
  let labe = add_instr (Emove(v0,retr,labs)) in
  add_lab_instr lexit (Egoto labe)
```

Traitement de la déclaration

```
type decl_fun =
 {name : string; params: int; entry:label; body : graph}
let ertl_fun {Rtl.name=id;Rtl.params=lr;Rtl.return=ret;
              Rtl.entry=lentry;Rtl.exit=lexit;Rtl.body=g}
 clear_graph(); ertl_graph g;
 let np = min (List.length lr) Register.nparams in
 let savers = List.map (fun r -> Register.fresh(),r)
                                  (ra::callee_saved)
 in
 let newentry = fun_entry savers lr lentry in
 fun_exit savers ret lexit;
 {name=id;params=np;entry=newentry;body = !graph}
```

- La compilation efficace de la récursion terminale devra se faire à ce stade

vers LTL: Location Transfer Language

- Les pseudo-registres sont associés à des registres physiques ou stockés sur la pile.
  - Ne pas mettre deux registres utiles simultanément dans le même registre physique (interférence)
  - Minimiser les opérations **move** en attribuant le même registre physique aux deux pseudo-registres (préférence)
- La taille des tableaux d'activation est déterminée
  - Suppression des instructions d'allocation et de desallocation de frame: arithmétique sur le pointeur $*sp*$
  - Les opérations d'accès aux paramètres deviennent des accès par rapport au sommet de pile.

Sommaire

Durée de vie

Définition

- Un registre est vivant sur un arc du graphe de flot de contrôle si sa valeur est utilisée avant d'être redéfinie.

Calcul

- Calcul d'une approximation : une variable pourra être déclarée vivante alors qu'elle n'est pas accédée dans les exécutions.
- Localement pour chaque instruction, on connait ce qui est défini et ce qui est utilisé

Calcul global sur le graphe de flot de controle

Analyse de flots de données

- Technique utilisée en analyse sémantique et en optimisation.
- Travail sur le graphe de flot de contrôle.
- À chaque point du graphe de flot on calcule certaines informations relatives au programme.
  - On distingue pour chaque nœud n, la valeur avant l'exécution de l'instruction $\texttt{in}(n)$ et celle après $\texttt{out}(n)$
  - Ces informations sont caractérisées par des équations locales:
    - $\texttt{in}(n)$ et $\texttt{out}(n)$ sont reliées en fonction de l'instruction associée au nœud.
    - $\texttt{out}(n)$ est relié à $\{\texttt{in}(p) | \text{arête de } n \text{ à } p\}$
  - Les boucles introduisent des dépendances circulaires
  - Un calcul de point fixe par itérations permet de gérer le calcul l'exécution.

Principes du calcul

- Approche en avant :
  - la sémantique de chaque instruction permet de définir $out(i)$ en fonction de $in(i)$ (fonction de transfert)
  - le flot de contrôle permet de définir $in(i)$ en fonction de $out(j)$ pour toutes les instructions j telles qu'il y a un arc de j à i.
- Approche en arrière :
  - la sémantique de chaque instruction permet de définir $in(i)$ en fonction de $out(i)$ (fonction de transfert)
  - le flot de contrôle permet de définir $out(i)$ en fonction de $in(j)$ pour toutes les instructions j telles qu'il y a un arc de i à j.

Sommaire

Exemple d'analyse en avant

Recherche des définitions actives

- Instructions $x = \ldots$ qui peuvent être utilisées pour une instruction.
- On stocke pour variable x, l'ensemble des couples (x, i) où i est une instruction qui définit x.
- Nombreuses applications: propagation de constantes, détection de variables non initialisées ...
- $out(i) = (x, i) \cup (in(i) - \{(x, k)\}_k)$ si $i \equiv x = d$
  $out(i) = in(i)$ si i ne définit aucune variable.
- $in(i) = \bigcup_{(j|j \to i)} out(j)$
  les définitions utiles avant i sont possiblement toutes les définitions utiles après les instructions qui mènent à i.
- On résoud ces équations par itération jusqu'à l'obtention d'un point fixe.

Exemple d'analyse en arrière

Recherche des variables vivantes

- Variables utilisées dans la suite sans être redéfinies.
- Utilisée pour l'allocation de registres.
- $in(i) = \texttt{used}(i) \cup (out(i) - \texttt{def}(i))$
- $out(i) = \bigcup_{(j|i \to j)} in(j)$
  les variables utiles après i sont les variables utiles avant toutes les instructions j qui peuvent suivre i.
- On résoud ces équations par itération jusqu'à l'obtention d'un point fixe.

Sommaire

Un théorème de point fixe

- Soit X un ensemble et F une fonction de X dans X.
- On cherche un point-fixe, c'est-à-dire p tel que $F(p) = p$

Remarques

- Une fonction arbitraire peut avoir 0 ou plusieurs points fixes
- Typiquement on aura $X = X_1 \times \cdots \times X_n$ lorsque l'on a n inconnues, par exemple les informations $\texttt{in}(i)$ et $\texttt{out}(i)$ pour chaque point de programme i.

Théorème de Tarski

- On suppose que X est un ensemble (partiellement) ordonné
- Tout sous-ensemble $Y \subseteq X$ a une borne inférieure ie il existe $x_0 \in X$ tel que (plus grand des minorants)
  1. $\forall y \in Y, x_0 \leq y$
  2. Si $\forall y \in Y, x \leq y$ alors $x \leq x_0$
- En particulier il existe un plus petit élément $\bot$ qui est la borne inférieure de l'ensemble X et un plus grand élément $\top$ qui est la borne inférieure de l'ensemble vide.

Exemples

- X est l'ensemble des parties d'un ensemble A avec l'ordre inclusion.
- Les booléens avec `false` $\leq$ `true`
- Ensemble D avec deux éléments $\bot$ et $\top$ et l'ordre $\bot \leq d \leq \top$.

Preuve

Théorème Toute fonction F monotone sur X admet un plus petit point fixe.

- Par hypothèse si $x \leq y$ alors $F(x) \leq F(y)$
- Soit $Y = \{y | F(y) \leq y\}$
- Soit p la borne inférieure de Y.
  - On montre : $F(p) \leq p$
    Il suffit de montrer que $\forall y \in Y, F(p) \leq y$ (borne-inf-2). Or si $y \in Y$, alors $p \leq y$ (borne-inf 1) donc $F(p) \leq F(y)$ (monotonie) et $F(y) \leq y$ (par déf de Y) donc finalement $F(p) \leq y$ (transitivité)
  - On montre $p \leq F(p)$
    Il suffit de montrer que $F(p) \in Y$ ie $F(F(p)) \leq F(p)$ (borne-inf (1)). On utilise la monotonie et la preuve précédente de $F(p) \leq p$.
  - Si y est un point fixe alors $F(y) \leq y$ dont $p \leq y$: p est le plus petit point fixe.

Autre théorème de point fixe

- On suppose que X est un ensemble (partiellement) ordonné
- Toute suite croissante $(x_n)_{n\in\mathbb{N}}$ a une borne supérieure ie il existe $x \in X$ tel que (plus petit des majorants)
  1. $\forall n, x_n \leq x$
  2. Si $\forall n, x_n \leq y$ alors $x \leq y$
- Il existe un plus petit élément $\bot$.
- On dit qu'une fonction monotone F est continue si pour toute suite croissante $(x_n)_{n\in\mathbb{N}}$, soit on a $F(\textbf{sup}\ (x_n)_n) = \textbf{sup}(F(x_n))_n$

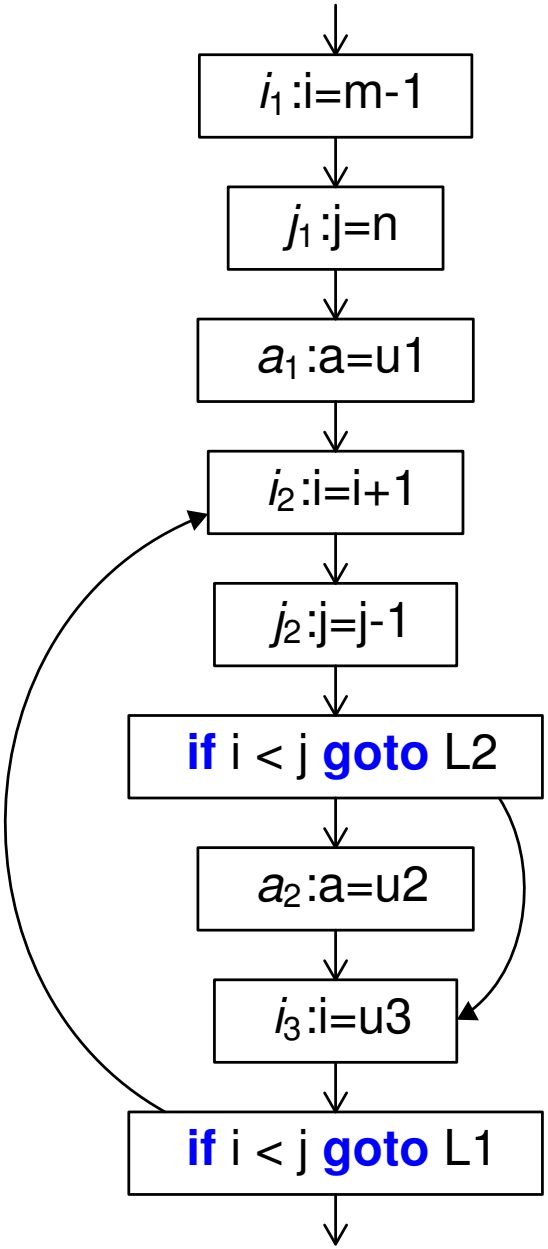
Théorème de point fixe Toute fonction croissante et continue admet un plus petit point fixe: $p = \textbf{sup}\ (F^n(\bot))_{n\in\mathbb{N}}$

- $\bot \leq F(\bot)$ dont $F^n(\bot) \leq F^{n+1}(\bot)$
- $F(p) = F(\textbf{sup}\ F^n(\bot)) = \textbf{sup}\ F^{n+1}(\bot) = \textbf{sup}\ F^n(\bot) = p$
- si y est un point fixe alors $\bot \leq y$ donc pour tout n, $F^n(\bot) \leq F^n(y) = y$ donc $p \leq y$

Exemple de calcul en avant: définitions active

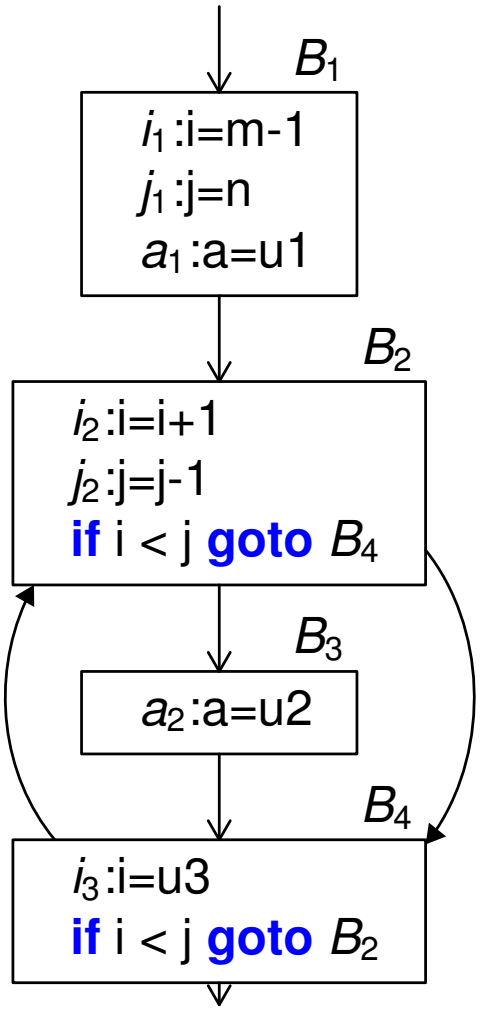
```
        i = m - 1
        j = n
        a = u1
L1:     i = i + 1
        j = j - 1
        if i < j goto L2
        a = u2
L2:     i = u3
        if i < j goto L1
```

Graphe de flots



$in(i_1)$	$=\emptyset$	$\emptyset$	$\emptyset$
$out(i_1)$	$=in(i_1)\backslash i \cup \{i_1\}$	i_1	i_1
$in(j_1)$	$=out(i_1)$	i_1	i_1
$out(j_1)$	$=in(j_1)\backslash j* \cup \{j_1\}$	i_1, j_1	i_1, j_1
$in(a_1)$	$=out(j_1)$	i_1, j_1	i_1, j_1
$out(a_1)$	$=in(a_1)\backslash a* \cup \{a_1\}$	i_1, j_1, a_1	i_1, j_1, a_1
$in(i_2)$	$=out(a_1) \cup out(if_2)$	i_1, j_1, a_1	$i_1, i_3, j_1, j_2, a_1, a_2$
$out(i_2)$	$=in(i_2)\backslash i* \cup \{i_2\}$	i_2, j_1, a_1	i_2, j_1, j_2, a_1, a_2
$in(j_2)$	$=out(i_2)$	i_2, j_1, a_1	i_2, j_1, j_2, a_1, a_2
$out(j_2)$	$=in(j_2)\backslash j* \cup \{j_2\}$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$in(if_1)$	$=out(j_2)$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$out(if_1)$	$=in(if_1)$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$in(a_2)$	$=out(if_1)$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$out(a_2)$	$=in(a_2)\backslash a* \cup \{a_2\}$	i_2, j_2, a_2	i_2, j_2, a_2
$in(i_3)$	$=out(a_2) \cup out(if_1)$	i_2, j_2, a_1, a_2	i_2, j_2, a_1, a_2
$out(i_3)$	$=in(i_3)\backslash i* \cup \{i_3\}$	i_3, j_2, a_1, a_2	i_3, j_2, a_1, a_2
$in(if_2)$	$=out(i_3)$	i_3, j_2, a_1, a_2	i_3, j_2, a_1, a_2
$out(if2)$	$=in(if2)$	i_3, j_2, a_1, a_2	i_3, j_2, a_1, a_2

Calcul par bloc



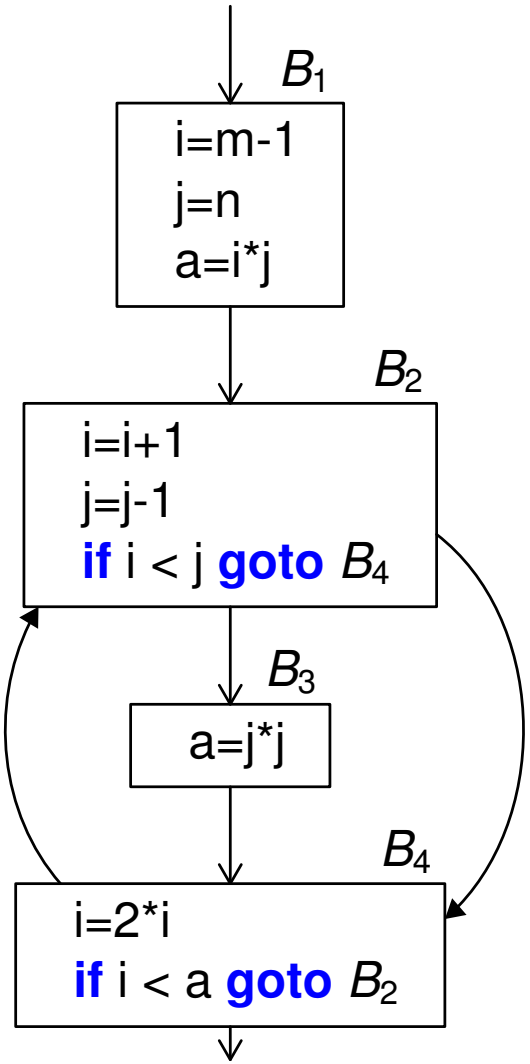
$in(B_1) = \emptyset$	$\emptyset$	$\emptyset$
$out(B_1) = in(B_1) \setminus \{i,j,a\} \cup \{i_1,j_1,a_1\}$	i_1, j_1, a_1	i_1, j_1, a_1
$in(B_2) = out(B_1) \cup out(B_4)$	i_1, j_1, a_1	$i_1, i_3, j_1, j_2, a_1, a_2$
$out(B_2) = in(B_2) \setminus \{i,j\} \cup \{i_2,j_2\}$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$in(B_3) = out(B_2)$	i_2, j_2, a_1	i_2, j_2, a_1, a_2
$out(B_3) = in(B_3) \setminus \{a\} \cup \{a_2\}$	i_2, j_2, a_2	i_2, j_2, a_2
$in(B_4) = out(B_3) \cup out(B_2)$	i_2, j_2, a_1, a_2	i_2, j_2, a_1, a_2
$out(B_4) = in(B_4) \setminus \{i\} \cup \{i_3\}$	i_3, j_2, a_1, a_2	i_3, j_2, a_1, a_2

Applications

- Recherche d'utilisations de variables non initialisées:
  - On ajoute une définition x_0 fictive après chaque déclaration de variable x.
  - Si lors d'une utilisation de x, la définition fictive x_0 est active, alors il y a un risque que x soit utilisée sans être initialisée.
- Propagation de constante
  - On cherche à repérer des définitions $x = d$ dans lesquelles d est une constante qui peut être calculée statiquement à la compilation.
  - Lors d'une définition $x = d$ on regarde les variables utilisées dans d.
  - La variable y est constante s'il y a une seule déclaration active et que cette déclaration est constante (s'il y a plusieurs déclarations actives, il faut qu'elles calculent la même valeur).
  - Si toutes les variables de d sont constantes alors la nouvelle déclaration $x = d$ est aussi constante.
  - Le compilateur doit calculer d. Le résultat peut dépendre de l'architecture.
  - On peut compiler le calcul de la constante et l'exécuter au vol.

Exemple d'analyse en arrière: variables vivantes

On suppose que *a* est utilisée dans le reste du programme (*in*(*fin*)={a}).



$in(B_1) = out(B_1)\setminus\{i,j,a\}\cup\{m,n\}$	m,n	m,n
$out(B_1) = in(B_2)$	a,i,j	a,i,j
$in(B_2) = out(B_2)\setminus\{i,j\}\cup\{i,j\}$	a,i,j	a,i,j
$out(B_2) = in(B_3)\cup in(B_4)$	a,i,j	a,i,j
$in(B_3) = out(B_3)\setminus\{a\}\cup\{j\}$	i,j	i,j
$out(B_3) = in(B_4)$	a,i	a,i,j
$in(B_4) = out(B_4)\setminus\{i\}\cup\{i,a\}$	a,i	a,i,j
$out(B_4) = in(B_2)\cup in(fin)$	a	a,i,j

Sommaire

Algorithme naif

- On initialise des tableaux (ou map) `in` et `out` avec la valeur minimale.
- On calcule les nouvelles valeurs dans `in'` et `out'` en utilisant les équations.
- Si `in'=in` et `out'=out` alors on s'arrête sinon on recommence à partir de `in'` et `out'`.

Optimisations

Cas d'une anlayse arrière

- modifier `in` et `out` en place économise de l'espace et accélère la convergence.
- garder trace dans un booléen de la modification d'une valeur pour poursuivre le calcul.
- choisir un ordre de calcul *intelligent*:
  "remonter" le graphe de flot de controle, calculer `out`(n) avant de calculer `in`(n)
- on peut aussi considérer un graphe de flot de contrôle par blocs
  - un bloc est une suite d'instructions consécutives toujours exécutées globalement
  - un bloc peut être considéré comme une macro-instruction
  - le graphe de flot de contrôle par bloc est sensiblement plus petit que le graphe par instruction
  - chaque bloc peut être ensuite optimisé

Algorithme de Kildall

- Lorsqu'une valeur `in`(n) change, cela peut affecter la valeur de `in`(p) pour les prédécesseurs p de n.
- On peut garder un ensemble de nœuds pour lesquels il faut vérifier que le point fixe est atteint.

Information attachée à chaque nœud de l'arbre

```
type live_info = {
  instr : instr;
  succ : Label.t list;
  mutable pred : Label.S.t;
  defs : Register.S.t;
  uses : Register.S.t;
  mutable ins: Register.S.t;
  mutable outs: Register.S.t;
}
```

`info` est une table qui contient pour chaque nœud du graphe une information de type `live_info`.

Calcul du point fixe

```
module RS = Register.S
let liveness g =
  let ws = ref Label.S.empty in
  Label.M.iter (fun l _ -> ws := Label.S.add l !ws) g;
  while not (Label.S.is_empty !ws) do
    let l = Label.S.choose !ws in
    ws := Label.S.remove l !ws;
    let li = Hashtbl.find info l in
    let outs = List.fold_left
        (fun outs s -> let lis = Hashtbl.find info s in
         RS.union outs lis.ins)
         RS.empty li.succ
    in
    let ins = RS.union li.uses (RS.diff outs li.defs) in
    if RS.cardinal ins > RS.cardinal li.ins
    then begin
      li.ins <- ins; li.outs <- outs;
      ws := Label.S.union li.pred !ws
    end
  done
```

Conclusion sur l'analyse de flots

- Une technique importante en compilation (analyse sémantique et génération de code)
- Une technique algorithmique générale sur les graphes qui a d'autres applications.

Prochaines étapes

- allocation de registres
- linéarisation du code