TP numéro 4 : Graphe de flot de contrôle

Assembleur – Compilation, ENSIIE

Semestre 3, 2024–25

L'objectif de ce TP est d'implémenter la transformation d'un AST en graphe de flot de contrôle. Pour simplifier, on se contentera d'un sous-ensemble de la syntaxe d'Untyped Pseudo Pascal décrit par les types suivants en OCaml :

```
type upp_expr =
  | Sub of upp_expr * upp_expr (* - *)
  | Li of int                  (* cte *)
  | Var of string
```

Une expression est une soustraction, une constante ou une variable.

```
type upp_cond =
  | Slt of upp_expr * upp_expr  (* set less than *)
  | Not of upp_cond
  | Or of upp_cond * upp_cond
```

Une condition est la comparaison stricte de deux expressions, ou la négation d'une condition, ou la disjonction de deux conditions.

```
type upp_instr =
  | Aff of string * upp_expr      (* s := e *)
  | Seq of upp_instr * upp_instr  (* i; i *)
  | IfThenElse of upp_cond * upp_instr * upp_instr
  | WhileDo of upp_cond * upp_instr
```

Une instruction est une affectation d'une expression dans une variable, ou une séquence de deux instructions, ou une conditionnelle, ou une boucle.

Un exemple de programme est fourni :

```
let euclid =
  Seq (Aff ("x", Li 42),
  Seq (Aff ("y", Li 24),
  WhileDo (Or (Slt (Var "x", Var "y"), Slt(Var "y", Var "x")),
           IfThenElse (Slt (Var "x", Var "y"),
                       Aff ("y", Sub (Var "y", Var "x")),
                       Aff ("x", Sub (Var "x", Var "y"))))))
```

qui correspond à la syntaxe concrète en Pseudo Pascal

```
x := 42;
y := 24;
while x < y or y < x do
  if x < y then
    y := y - x
  else
    x := x - y
```

Les nœuds d'un graphe RTL seront quand à eux représentés par le type

```
type rtl_node = {
  label : label;
  op : string;
  dest : pseudo_register;
  args : arg list;
  exits : label list;
}
```

où les `label`s sont des `string`s, les `pseudo_register`s des `int`s, et les arguments `arg` sont définis par le type

```
type arg = Cte of int | Reg of pseudo_register
```

Ainsi, `l1: sub %1, %2, %3 -> l2` sera représenté par

```
{ label = "l1"; op = "sub"; dest = 1;
  args = [Reg 2; Reg 3]; exits = ["l2"] }
```

tandis que `l3: bgtz %4 -> l4, l5` sera représenté par

```
{ label = "l3"; op = "bgtz"; dest = 4;
  args = []; exits = ["l4"; "l5"] }
```

Un programme RTL est donc une liste de tels nœuds.
On dispose de fonctions

```
val fresh_pseudo_register : unit -> pseudo_register

val fresh_label : unit -> label
```

qui permettent de créer des pseudo-registres et des étiquettes fraîches, ainsi que d'une table de hachage

```
val id_to_pseudo_register_map : (string, pseudo_register) Hashtbl.t
```

qui fait le lien entre les noms de variables et les pseudo-registres. Cette table est initialisée avec des pseudo-registres frais pour les variables `"x"` et `"y"`. Pour trouver l'association d'un identifiant `id`, il faut donc faire

```
Hashtbl.find id_to_pseudo_register_map id
```

Enfin, on dispose de fonctions d'affichage

```
val pp_rtl : Format.formatter -> rtl -> unit

val dot_rtl : Format.formatter -> rtl -> unit
```

qui affiche un programme RTL en syntaxe concrète, ou sous forme d'un graphe orienté qu'il est possible de passer au programme `dot`. La fin du fichier fourni vous montre des exemples d'utilisation possible. Une fois le TP terminé, il devrait être possible de taper

```
> ./mon_programme | dot -Tx11
```

pour afficher le graphe de flot de contrôle du programme `euclid`.

1. Implémenter une fonction

   ```
   val expr_to_rtl :
     label -> label -> pseudo_register -> upp_expr -> rtl -> rtl
   ```

   telle que `expr_to_rtl in_label out_label reg e rtl` complète le graphe `rtl` en y ajoutant les nœuds pour calculer l'expression `e` de telle façon que ce calcul commence en `in_label`, sorte en `out_label` et place le résultat calculé dans `reg`.

   Remarque : dans le cas d'une variable, il faudra donc faire un `move` depuis le registre correspondant à la variable, vers `reg`.

2. Implémenter une fonction

   ```
   val cond_to_rtl :
     label -> label -> pseudo_register -> upp_cond -> rtl -> rtl
   ```

   telle que `cond_to_rtl in_label out_label reg c rtl` complète le graphe `rtl` en y ajoutant les nœuds pour calculer la condition `c` de telle façon que ce calcul commence en `in_label`, sorte en `out_label` et place le résultat calculé dans `reg`.

   On fera attention à bien respecter le comportement coupe-circuit du `or`.

3. Implémenter une fonction

   ```
   val instr_to_rtl :
     label -> label -> upp_instr -> rtl -> rtl
   ```

   telle que `instr_to_rtl in_label out_label i rtl` complète le graphe `rtl` en y ajoutant les nœuds pour calculer l'instruction `i` de telle façon que ce calcul commence en `in_label` et sorte en `out_label`.