PROJET COMPILATION

GROUPE LA TRUITE OPAQUE V2

DESBRUS Quentin
BLETON--GIORDANO Maxence

DRIOWYA Abdelghafour
KOSSONOGOW Justin

ORGANISATION DU GROUPE

- Attribution des tâches
- Gestion de version
- Communication

COMMUNICATION

	BOGUE	DATE DE CRÉATION	CESSIONNAIRE	DATE LIMITE	ETAT	GRAVITÉ
CM1-I18	Allocation de registre difficile (à la fin)	12-17-2018	Justin	-	Ouvert	Aucun
CM1-I17	Automatiser les tests	12-17-2018	Maxence	-	In progress	Aucun
CM1-I16	2/backend (écrire les tests)	12-16-2018	quentin	-	In progress	Aucun
CM1-I15	2/frontend (écrire les tests)	12-16-2018	Justin	-	In progress	Aucun
CM1-I14	Documentation (à la fin)	12-16-2018	Non affec...	-	Ouvert	Aucun
CM1-I13	2/Typage (écrire les tests)	12-15-2018	Moi	-	In progress	Aucun

TYPE CHECKING

- Etapes :
  - Génération d'une liste d'équations avec un visiteur
  - Résolution de la liste d'équations avec une méthode récursive
- Exemples :
  - Valide (add) : `print_int (1+1)`
  - Invalide(add) : `print_int (1+true)`
  - Valide (fonction) : `let rec f x = x+1 in print_int (f 15)`
  - Invalide (fonction) : `let rec f x = f in ()`
    - En effet, on a type(fonction) = type(paramètre)->type(résultat) et si la fonction se renvoie elle-même, le type de son résultat est égal au type de la fonction.
    - Donc type(résultat)=type(fonction) = type(paramètre)->type(résultat)
    - Et type(résultat) = type(paramètre)->type(résultat) est absurde
    - Donc le programme est mal typé
  - Valide(tableau) : `let x = Array.create 1 2 in print_int (x.(0))`
  - Invalide(tableau) : `let x = Array.create true () in print_int (x.(0))`

K NORMALISATION

Arbre MinCaml

```
print_int (1 + 2 + 3)
```

Arbre MinCaml après cette étape

```
(let v1 =
   (let v3 = 3 in
      (let v2 = (let v5 = 2 in (let v4 = 1 in (v4 + v5))) in (v2 + v3)))in
         (let v0 = print_int in (v0 v1)))
```

- Permet de faire en sorte que les opérandes soient uniquement dans des variables.
- Produit un arbre MinCaml où les opérateurs arithmétiques n'ont que deux opérandes.

ALPHA CONVERSION

Arbre MinCaml

```
let x = 14 in
     let x = 10 in
         print_int(x)
```

Arbre MinCaml après cette étape

```
(let v2 = 14 in
     (let v3 = 10 in
          (print_int v3)))
```

- Renommage des variables pour simplifier certaines étapes comme l'inline expansion.

BETA REDUCTION

Arbre MinCaml

```
let x = 14 in
    let y = x in
        print_int(y)
```

Arbre MinCaml après cette étape

```
(let x = 14 in
    (let y= x in
        (print_int x)))
```

INLINE EXPANSION

Arbre MinCaml

```
let rec afficher z = print_int(z) in
    let x = 5 in
        afficher x
```

Arbre MinCaml après cette étape

```
(let rec afficher z =(print_int z) in
        (let x = 5 in
                (print_int x)))
```

- Remplace les appels de fonctions par leur corps si son nombre de nœud ne dépasse pas une taille maximale (15 dans notre cas).

REDUCTION DES LET IMBRIQUÉS

Arbre MinCaml

```
let x =  (let y = 10 in y)
in
    print_int(x)
```

Arbre MinCaml après cette étape

```
( let y = 10 in
    (let x = y in
        (print_int x)))
```

- Pour un nœud de la forme let x = e1 in e2, on élimine les lets imbriqués dans e1 et e2, puis on utilise une méthode récursive qui, si e1 est un nœud let, le place avant l'autre nœud let.

CONSTANT FOLDING

Arbre MinCaml

```
let x =  7 in
    let y = 2 in
          let z = x + y in print_int(z)
```

Arbre MinCaml après cette étape

```
(let x = 7 in
    (let y = 2 in
        (let z = 9 in
            (print_int z))))
```

- On ne propage pas les constantes dans certaines situations (pour les paramètres de fonctions par exemple) pour éviter de devoir défaire cette propagation de constante pour la traduction en ASML

SUPPRESSION DES DÉFINITIONS INUTILES

Arbre MinCaml

```
let x =  7 in
    let y = 2 in
         let z = 5 in print_int(z)
```

Arbre MinCaml après cette étape

```
   (let z = 5 in
        (print_int z))
```

- Pour nœud de la forme let x = e1 in e2, si la variable déclarée n'est pas utilisée dans e2 et qu'il n'y pas d'effet de bord dans e1 (pas d'appel de fonction ni d'écriture dans un tableau) on remplace le nœud let par e2.

GENERATION DE L'ARBRE ASML

Arbre MinCaml

```
let x =  9 in
    let y = 1 + x in
         print_int(y)
```

Arbre ASML généré

```
let _ =
          let v4 = 9
          in
          let v7 = 1
          in
          let v5 = add v7 v4
          in
          call_min_caml_print_int v5
```

- Après les étapes précédentes, l'arbre MinCaml est déjà très proche d'un arbre ASML. Dans cet exemple, le constant folding a été désactivé pour pouvoir montrer comment est traité un noeud add dans un programme court)

CLOSURES

Fonction avec une variable libre (a)

```
let a = 1 in
   let rec _f x = a in
      _f 0
```

Fonction utilisée autrement qu'en l'appelant

```
let rec _f x = x in
    let h = _f in
       h 1
```

Fonction _f utilisant une closure (_g en est une)

```
let a = 1 in
let rec _g x = x + a in
let rec _f x = _g x in
    f 2
```

- Si au moins une de ces 3 conditions est vérifiée, il faut utiliser une closure pour cette fonction (appelée _f dans les 3 exemples)

CLOSURES

Arbre MinCaml

```
let a = 1 in
   let b = 2 in
      let rec _f x =
         if x = 0 then a+b else f 0
      in _f 1
```

Arbre ASML généré

```
let _f x =
    let a = mem(%self+1) in
    let b = mem(%self+2) in
    let zero = 0 in
    if x = zero then add a b else call_closure %self zero

Let _ =
    let a2 = 1 in
    let b2 = 2 in
    let un = 1 in
    let closureF = new 12 in
    let adresseF = _f in
    let ecriture1 = mem(closureF + 0) <- adresseF in
    let ecriture2 = mem(closureF + 1) <- a2 in
    let ecriture3 = mem(closureF + 2) <- b2 in
    call_closure closureF un
```

- Pour chaque closure, il faut allouer une zone mémoire (**closureF** dans cet exemple) et écrire l'adresse de la fonction et ses variables libres dans cette zone. Lors de son appel, la fonction peut accéder à cette zone mémoire avec l'identificateur spécial **%self**

ALLOCATION DE REGISTRES

Arbre ASML

```
let _ =
    let x = 1 in
       if x = 1
       then
           let y = 1 in
           y
       else
           let z = 1 in
           x + z
```

Spill everything

- x est stocké à l'adresse FP
- y est stocké à l'adresse FP-4
- z est stocké à l'adresse FP-4

Linear scan

- x est stocké dans R6
- y est stocké dans **R8**
- z est stocké dans R8

Tree scan

- x est stocké dans R6
- y est stocké dans **R6**
- z est stocké dans R8

ALLOCATION DE REGISTRES

Pour un nœud de la forme let x = e1 in e2, l'algorithme linear scan libère les registre alloués à une variable qui n'est pas utilisée dans e2 uniquement si la variable a été déclarée dans la même suite de let que ce noeud.

```
let x1 = a1 in
let x2 = a2 in
let x3 = a3 in
…
Let xn = an in
in b
```

Une suite de let en ASML

```
let _ =
    let x = 1 in
    let a = if x = 1
            then
               let y = 1 in
               y
            else
               let z = 1 in
               x + z
    in 1
```

Exemple d'arbre ASML ou le tree scan effectue une meilleure allocation de registre que le linear scan

```
let _ =
    let x = 1 in
    let a = if x = 1
            then
               let y = 1 in
               y
            else
               let z = 1 in
               x + z
    in x
```

Exemple d'arbre ASML ou le tree scan effectue la même allocation de registre que le linear scan

GENERATION DE CODE ARM

Noeud add avec des variables stockées sur la pile :

Code ASML :

```
let _ =
    ...
    in
    let v4 = add v6 v5
    in
    ...
```

Code ARM généré :

```
    ...
    LDR R4, [FP, #-4]
    LDR R5, [FP, #0]
    ADD R4, R4, R5
    STR R4, [FP, #-8]
    ...
```

Noeud add avec des variables stockées dans les registres:

Code ASML :

```
let _ =
    ...
    in
    let v4 = add v6 v5
    in ...
```

Code ARM généré :

```
    ...
    ADD R8, R9, R10
    ...
```

GENERATION DE CODE ARM

- Exemple d'appel de fonction

  Code ASML :

```
let _ =
   let v2 = 42
   in
   call _min_caml_print_int v2
```

Code ARM généré :

```
PUSH {R4, R5, R6, R7, R8, R9, R10, FP}
SUB FP, SP, #4
LDR R8, =42
PUSH {R0, R1, R2, R3, R12, LR}
MOV R0, R8
BL min_caml_print_int
MOV R7, R0
POP {R0, R1, R2, R3, R12, LR}
MOV R0, R7
POP {R4, R5, R6, R7, R8, R9, R10, FP}
B min_caml_exit
```

GENERATION DE CODE ARM

Noeud fadd avec des variables stockées sur la pile :

Code ASML :

```
let _ =
    ...
    let v4 = fadd v6 v5
    in
    ...
```

Code ARM généré :

```
    ...
    FLDS S0, [FP, #-4]
    FLDS S1, [FP, #0]
    FADDS S0, S0, S1
    FSTS S0, [FP, #-8]
    ...
```

Noeud fadd avec des variables stockées dans les registres:

Code ASML :

```
let _ =
    ...
    let v4 = fadd v6 v5
    in
    ...
```

Code ARM généré :

```
    ...
    FMSR S0, R8
    FMSR S1, R9
    FADDS S0, S0, S1
    FMRS R10, S0
    ...
```