Machines Virtuelles – MV6
CM 5

Peter Habermehl, Merci à Michele Pagani et ...

Université Paris Cité
UFR Informatique
IRIF

Peter.Habermehl@irif.fr

14 mars 2024

Compilation des fonctions: les fermetures (clôtures, closures)

Rappel

- En OCaml, on a deux "opérations" d'ordre supérieur:

  `fun x -> e1` : des expressions de type "fonction"

  `e1 e2` : un opérateur d'application

- et du sucre syntactique:

```
let f x y = x+y in f 2 4
```

  qui est vraiment

```
let f = (fun x -> fun y -> x+y) in f 2 4
```

- les fonctions sont des expressions de première classe

```
let g x = fun f -> f(f x) in g (fun x -> x+1)
```

```
let succ = (+) 1 in succ 3
```

Fonctions $\mapsto$ closure + code

- Compilation de

  ```
  fun x -> x + 1
  ```

- On veut une suite d'instructions telle que:
  - à la fin de l'exécution, l'accumulateur contienne le résultat
  - le résultat est la fonction.
- On utilise un bloc. L'accumulateur est

1b 0	31b 0x0143

- un pointeur sur un bloc avec ici une seule donnée, dans le tas:

  0x0143 $\mapsto$

32b entête :		
8b 247	2b (color)	22b 1

32b		
1b 0	31b position dans le code	

Fonctions $\mapsto$ closure + code

Compilation de

```
fun x -> x + 1
```

avec `ocaml -dinstr`:

```
    closure L1, 0
    return 1
L1: acc 0
    offsetint 1
    return 1
```

Les adresses dans le code sont symbolisées par des labels.

Fonctions $\mapsto$ closure + code

Les instructions:

`closure L1, 0` crée une fermeture de label L1 (un bloc de tag 247), puis met son adresse dans `l'accu`.

`acc 0` f accède à son premier argument: il est sensé être rangé sur la pile.

`return 1` f "rend la main", en dépilant 1 case de la pile; sa valeur de retour est lue dans `l'accu`.

On a aussi:

`apply 1` lance l'exécution de f (`l'accu` contient la fermeture); son argument est au sommet de `S`, plus autre chose: voir description de `apply1` (!) dans la doc

Un exemple au tableau !

Compilation de

```
let f x = x+1 in 2*(f 4)
```

avec `ocaml -dinstr`

```
    closure L1, 0
    push
    const 4
    push
    acc 1
    apply 1
    push
    const 2
    mulint
    return 2
L1: acc 0
    offsetint 1
    return 1
```

D'où repartir ?

- Où repartir à la fin d'un appel de fonction (après `return`) ?
- Besoin de mémoriser PC lors de l'appel:

  `apply 1` sauve (entre autres) le PC (+1) sur la pile (en dessous de l'argument)

  `return 1` s'attend (entre autres) à trouver sur la pile un PC à restaurer (après avoir enlevé un élément de la pile)

Un exemple plus délicat...

Compilation de:

```
let a = 1 in
let f x = x+a in 2*(f 4)
```

avec `ocaml -dinstr`

```
const 1
push
...
closure L1, ...
push
const 4
push
acc 1
apply 1
push
const 2
mulint
return 3
```

```
L1: ???     /*compilation de a
    push
    acc 1   /*compilation de x
    addint
    return 1
```

Où mettre les valeurs des variables libres ?

Problème:

- La fonction `f` utilise une variable `a` défini en dehors de son code.
- Comment y accéder ?

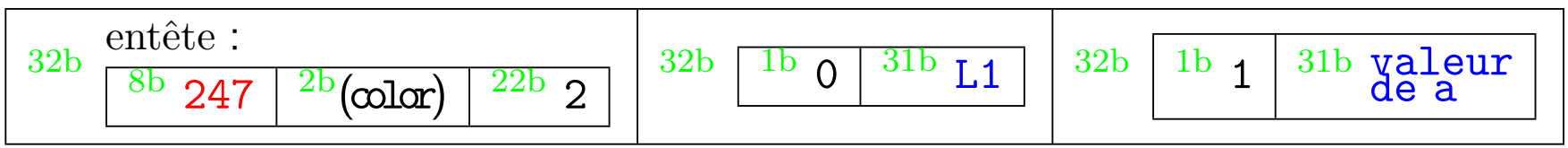
Où mettre les valeurs des variables libres ?

Problème:

- La fonction `f` utilise une variable `a` défini en dehors de son code.
- Comment y accéder ?

Solution:

- Stocker dans la fermeture les valeurs de toute variable utilisée dans la fonction courante:



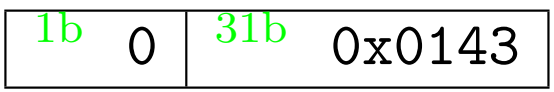
- Ajouter une instruction qui permet d'accéder aux cases de cette fermeture
  - utiliser registre `env` de ocamlrun qui pointe toujours sur la fermeture de la fonction courante

Nouvelles instructions

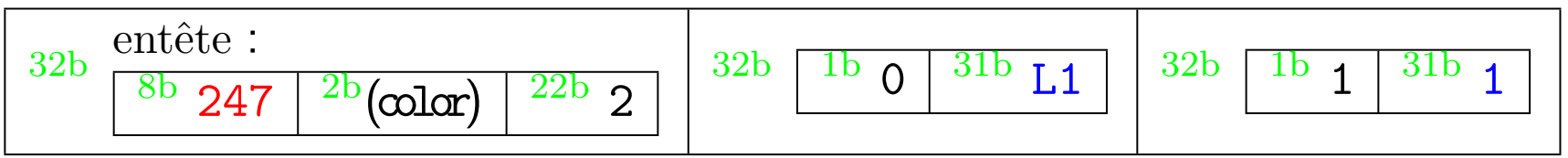
`envacc n` met dans `l'accu` le contenu de la (n-1)-ième case de l'environnement courant (pointé par `env`).

`closure L, n` met dans `l'accu` un pointeur vers une fermeture avec n éléments dans son environnement, prises dans `l'accu` et dans la pile `S`.

Donc la fermeture de `f` est



Le pointeur dans le tas: `0x0143` $\mapsto$



Un exemple plus délicat...

Compilation de:

```
let a = 1 in
let f x = x+a in 2*(f 4)
```

avec `ocaml -dinstr`

```
const 1
push
acc 0
closure L1, 1
push
const 4
push
acc 1
apply 1
push
const 2
mulint
return 3
```

```
L1: envacc 2 /*compilation de a
    push
    acc 1   /*compilation de x
    addint
    return 1
```

Exercice 1

Que fait le code suivant ? Quelle expression l'a généré ?

```
const 1
push
const 2
push
acc 0
push
acc 2
closure L1, 2
push
const 3
push
const 4
push
acc 2
apply 1
push
acc 1
mulint
return 5
```

```
L1: acc 0
    push
    envacc 3
    push
    envacc 2
    addint
    addint
    return 1
```

Exercice 1

Que fait le code suivant ? Quelle expression l'a généré ?

```
const 1
push
const 2
push
acc 0
push
acc 2
closure L1, 2
push
const 3
push
const 4
push
acc 2
apply 1
push
acc 1
mulint
return 5
```

```
L1: acc 0
    push
    envacc 3
    push
    envacc 2
    addint
    addint
    return 1
```

Solution:

```
let a = 1 in
let b = 2 in
let f x = a+b+x in
let c = 3 in
 c*(f 4)
```

Exercice 2

Que fait le code suivant ? Quelle expression l'a généré ?

```
closure L2, 0
push
acc 0
closure L1, 1
push
const 2
push
acc 1
apply 1
push
const 4
addint
return 3
```

```
L1: acc 0
    push
    envacc 2
    apply 1
    push
    const 3
    addint
    return 1

L2: acc 0
    offsetint 1
    return 1
```

Exercice 2

Que fait le code suivant ? Quelle expression l'a généré ?

```
 closure L2, 0
 push
 acc 0
 closure L1, 1
 push
 const 2
 push
 acc 1
 apply 1
 push
 const 4
 addint
 return 3
```

```
L1: acc 0
    push
    envacc 2
    apply 1
    push
    const 3
    addint
    return 1

L2: acc 0
    offsetint 1
    return 1
```

Solution:

```
let f x = x+1 in let g x = 3 + (f x) in
 4 + (g 2)
```

Compilation des fonctions récursives

La récursion

```
let rec f x = 1 + f x in 2 * (f 3)
```

Un appel récursif se fait:

- en chargeant dans `l'accu` la fermeture courante contenue dans `env`
  $\Rightarrow$ autoreference !
- en l'appliquant comme précédemment.

Instructions.

`offsetclosure 0` copie le contenu de `env` dans `l'accu`

`closurerec k, n` comme `closure Lk, n; push`

La récursion

```
let rec f x = 1 + f x in 2 * (f 3)
```

```
closurerec 1, 0
const 3
push
acc 1
apply 1
push
const 2
mulint
return 2
```

```
L1: acc 0
    push
    offsetclosure 0
    apply 1
    push
    const 1
    addint
    return 1
```

La récursion: exercice

```
closurerec 1, 0
const 2
push
acc 1
apply 1
push
const 1
addint
return 2
```

```
L1: acc 0
    push
    const 0
    eqint
    branchifnot L2
    const 1
    return 1
L2: acc 0
    offsetint -1
    push
    offsetclosure 0
    apply 1
    push
    acc 1
    mulint
    return 1
```

La récursion: exercice

```
closurerec 1, 0
const 2
push
acc 1
apply 1
push
const 1
addint
return 2
```

Solution:

```
let rec fact x =
if x=0 then 1
else x*(fact (x-1))
in 1+(fact 2)
```

```
L1: acc 0
    push
    const 0
    eqint
    branchifnot L2
    const 1
    return 1
L2: acc 0
    offsetint -1
    push
    offsetclosure 0
    apply 1
    push
    acc 1
    mulint
    return 1
```

La récursion mutuelle

```
let rec f x = 1 + g x
and g x = 3 * h x
and h x = if x=0 then 2 else f (x-1)
in f 2
```

Pour fonctions récursives mutuelles: une même fermeture pour tout un paquet mutuel:

hdr	x	pc + o_1	hdr	x	pc + o_2	...hdr	x	pc + o_k	e_1	...	e_n

Instructions.

`closurerec` $o_1, \ldots o_k$, `n` fabrique la fermeture ci-dessus, en prenant `n` éléments d'environnement e_1, ..., e_n dans `l'accu+pile`:

`offsetclosure n` copie le contenu de `env+n` dans `l'accu`

```
let rec f x = 1 + g x
    and g x = 3 * h x
and h x = if x=0 then 2 else f (x-1)
                        in f 2
```

```
closurerec 1 2 3, 0
const 2
push
acc 3
appterm 1, 5
```

```
L1: acc 0
    push
    offsetclosure 3
    apply 1
    push
    const 1
    addint
    return 1

L2: acc 0
    push
    offsetclosure 3
    apply 1
    push
    const 3
    mulint
    return 1
```

```
L3: acc 0
    push
    const 0
    eqint
    branchifnot L4
    const 2
    return 1

L4: acc 0
    offsetint -1
    push
    offsetclosure -6
    appterm 1, 2
```

Optimisations

Exemples d'optimisation

- L'encodage des fonctions peut être amélioré sur plusieurs endroits:
  - applications n-aires
  - récursion mutuelle
  - élimination des filtrages
  - appels terminaux
  - ...
- Nous détaillerons deux cas particuliers.

Exemples d'optimisation

- L'encodage des fonctions peut être amélioré sur plusieurs endroits:
  - applications n-aires
  - récursion mutuelle
  - élimination des filtrages
  - appels terminaux
  - ...
- Nous détaillerons deux cas particuliers.

Applications n-aires

`fun x -> fun y -> x+y`
devrait donner deux fermetures:

```
    closure L1, 0
    return 1
L2: acc 0
    push
    envacc 2
    addint
    return 1
L1: acc 0
    closure L2, 1
    return 1
```

```
    closure L1, 0
    return 1
    restart
L1: grab 1
    acc 1
    push
    acc 1
    addint
    return 2
```

- à la place, on la considère comme une fonction à deux arguments, `x` et `y` et en fonction du nombre d'arguments fourni à l'appel, on effectue des opérations différentes.

Applications n-aires

`fun x -> fun y -> x+y`
devrait donner deux fermetures:

```
    closure L1, 0
    return 1
L2: acc 0
    push
    envacc 2
    addint
    return 1
L1: acc 0
    closure L2, 1
    return 1
```

```
    closure L1, 0
    return 1
    restart
L1: grab 1
    acc 1
    push
    acc 1
    addint
    return 2
```

- à la place, on la considère comme une fonction à deux arguments, `x` et `y` et en fonction du nombre d'arguments fourni à l'appel, on effectue des opérations différentes.

Applications n-aires

On utilise le registre `extra_args` de ocamlrun: il stocke le nombre d'arguments prêts pour être appliqués.

`apply1...3` `l'accu` contient une fermeture, le sommet de `S` contient n arguments puis une zone où sont sauvés PC(+1)/`env`/`extra_args`.

- `env` reçoit `l'accu`
- `extra_args` reçoit $n-1$
- `pc` saute au label de la fermeture pointé par `l'accu`

`apply n` pour $n > 3$ idem mais sans sauvegarde de PC/`env`/`extra_args` qui doit être fait par `push-retaddr`

Applications n-aires

`grab n` teste si une fonction n-aire a eu assez d'arguments, deux cas:

- si `extra_args` $\geq n$, on continue avec `extra_args` décrémenté de n et exécute la suite
- sinon avant de retourner sans exécuter la suite on place les (`extra_args`+1) arguments présents sur la pile dans une fermeture de la forme:

Header	Lbl de `RESTART`	pointeur env	arg0	...	argk

on récupère l'adresse de retour de la pile et on restaure `extra_args` et `env`.

`restart` `env` doit pointer vers une fermeture de la forme précédente. On replace alors les arguments sur la pile, et on incrémente `extra_args` d'autant et on récupère l'environnement de la fermeture.

```
let f x y z = z+x+y in
let g = f 1 in
let h = g 2 in
1+h 3
```

```
closure L1, 0
push
const 1
push
acc 1
apply 1
push
const 2
push
acc 1
apply 1
push
const 3
push
acc 1
apply 1
push
const 1
addint
return 4
```

```
    restart
L1: grab 2
    acc 1
    push
    acc 1
    push
    acc 4
    addint
    addint
    return 3
```

Appels terminaux

```
let f x = x+1 in 2*(f 4)
```

```
    closure L1, 0
    push
    const 4
    push
    acc 1
    apply 1
    push
    const 2
    mulint
    return 2
L1: acc 0
    offsetint 1
    return 1
```

```
let f x = 2*(x+1) in f 4
```

```
    closure L1, 0
    push
    const 4
    push
    acc 1
    appterm 1, 3
L1: acc 0
    offsetint 1
    push
    const 2
    mulint
    return 1
```

Appels terminaux

- appel terminal: une fonction g appelle une fonction f, mais le résultat envoyé par f est envoyé tout de suite par g
- on n'a plus besoin des valeurs locales de la fonction g quand on lance f
- l'espace sur la pile S peut être libérée avant de lancer f

Instruction.

`appterm n, s` garde les premières n éléments de la pile S (les arguments) et efface les autres $s - n$ (les valeurs locales de g), ensuite appel la fonction dont l'adresse se trouve dans `l'accu`.

- Intérêt: une fonction récursive dont tous les appels sont terminaux s'effectuera en pile constante (comme une boucle).

```
                                fun x -> let rec aux x a =
                            if x = 0 then a else aux (x-1) (x*a)
                                                        in aux x 1
closure L1, 0                    restart            L1: closurerec 2, 0
return 1                    L2:  grab 1                 const 1
                                 acc 0                  push
                                 push                   acc 2
                                 const 0                push
                                 eqint                  acc 2
                                 branchifnot L3         appterm 2, 4
                                 acc 1
                                 return 2
                            L3:  acc 1
                                 push
                                 acc 1
                                 mulint
                                 push
                                 acc 1
                                 offsetint -1
                                 push
                                 offsetclosure 0
                                 appterm 2, 4
```