
Introduction aux machines virtuelles

Cours 3 : La Machine Virtuelle Caml, suite

Pierre Letouzey

Prenom.Nom@pps.jussieu.fr

Mémoire et données Caml

Toute donnée Caml correspond à **un** mot machine (32/64 bits):

- soit cet entier est directement la valeur (`int`, `bool`,...)
- soit cet entier est un pointeur vers un **bloc** mémoire

Données codés par des entiers

- `int`: entiers signés sur 31 ou 63 bits (cf plus tard)
- `char`: codé par le code ASCII correspondant
- `bool`: `false` (resp. `true`) est codé par 0 (resp. 1)
- `unit`: `()` est codé par 0
- Dans un type concret `type t = A | B of int | C`
les constructeurs constants `A`, `C` sont des numéros: 0 et 1.
- En particulier: `[]` est codé par 0, tout comme `None`, ...

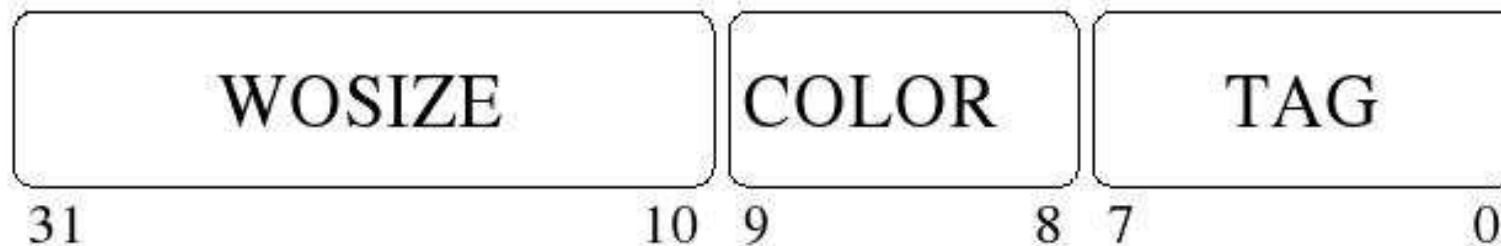
Données allouées par blocs : tout le reste !

Un bloc en mémoire : une portion du **tas** (ou **heap**).

L'entier référençant cette zone est juste son adresse (pointeur).

Le bloc commence par un entier d'entête, suivi des données

Entête (sur une machine 32-bits):



Exemple de Tag (cf `byterun/mlvalue.h`):

- Pour un `float`: 253
- Pour une fonction (clôture): 247
- $\text{Tag} \in [0..245]$: un constructeur non-constant

Exemples de blocs de données allouées

- Un tableau : [|1;2|]

Wosize = 2

Color = ?? (initialement white, soit 0)

Tag = 0 (1er “constructeur” du type array)

En mémoire:

$2 * 2^{10}$	1	2
--------------	---	---

(en fait pas tout à fait exact, cf plus loin)

- Une paire : (1,2)

Même représentation en mémoire !!

Démo: modification magique d'une paire en tableau

- Une liste : [1;2]

Deux blocs de taille 2 (Cf. dessin au tableau) :

$2 * 2^{10}$	1	addr_2nd_bloc
--------------	---	---------------

$2 * 2^{10}$	2	0
--------------	---	---

La séparation entre entiers et pointeurs

Tous les blocs alloués peuvent un jour devenir inutiles:

Un Garbage Collector (GC) fait parfois le ménage.

⇒ Besoin de **parcourir** les données (cf Color).

⇒ Besoin de savoir reconnaître rapidement les pointeurs.

Convention Caml: Tout nombre pair est un pointeur,

Tout nombre impair $2n + 1$ représente en fait n .

C'est la raison du bit manquant dans la taille de `int` !

Retour sur les exemples précédents:

- 0 est codé en mémoire par 1
- 1 par 3, etc

Instructions de la V.M. pour les blocs

- **MAKEBLOCK** n k : fabrique un bloc de tag k et de données n éléments retirés du haut de **accu+pile**.
accu reçoit ensuite le pointeur vers le bloc ainsi créé.
- **GETFIELD** n : **accu** doit contenir un pointeur vers un bloc.
accu reçoit alors la n -ième donnée du bloc.
- **SETFIELD** n : **accu** doit contenir un pointeur vers un bloc.
La n -ième donnée du bloc est alors remplacée par la valeur prise au somme de la **pile**. **accu** reçoit ensuite $()$.
- **VECTLENGTH**: **accu** reçoit le nombre de données du bloc pointé par **accu**.

Instructions de la V.M. pour les blocs: variantes

- MAKEBLOCK1,2,3, GET/SETFIELD0,1,2,3 : cf. avant.
- GETVECTITEM, SETVECTITEM: cf GETFIELD et SETFIELD, mais avec tous les arguments dans **accu+pile**.
- ATOM k : remplace MAKEBLOCK 0 k dans le cas d'un bloc avec 0 données (essentiellement [| |]). Il existe aussi: ATOM0, PUSHATOM k et PUSHATOM0.
- Les tableaux de float sont traités à part: MAKEFLOATBLOCK n, GET/SETFLOATFIELD n
- Les chaînes de caractères aussi: GET/SETSTRINGCHAR
- OFFSETREF n : ajoute n au premier champ d'un bloc.

Instructions de la V.M. : switch

Cette instruction va permettre de réaliser le match de Caml:

`switch` $l_0^c \dots l_n^c / l_0^{nc} \dots l_m^{nc}$: on regarde le contenu de **accu**:

- S'il s'agit d'un nombre impair $2 * i + 1$, il représente en fait l'entier i , et plus précisément le i -ème constructeur constant. On effectue donc le déplacement indiqué par l_i^c
- S'il s'agit au contraire d'un pointeur, on est en présence d'un constructeur non-constant. On accède au bloc via le pointeur. Si le tag présent dans l'entête du bloc est i , on effectue alors le déplacement l_i^{nc} .

Les fonctions

Suite et fin de la partie Caml la séance suivante. Reste à voir:

APPLY

APPTERM

CLOSURE

CLOSUREREC

RETURN

RESTART

GRAB

PUSH_RETADDR

(PUSH) OFFSETCLOSURE