

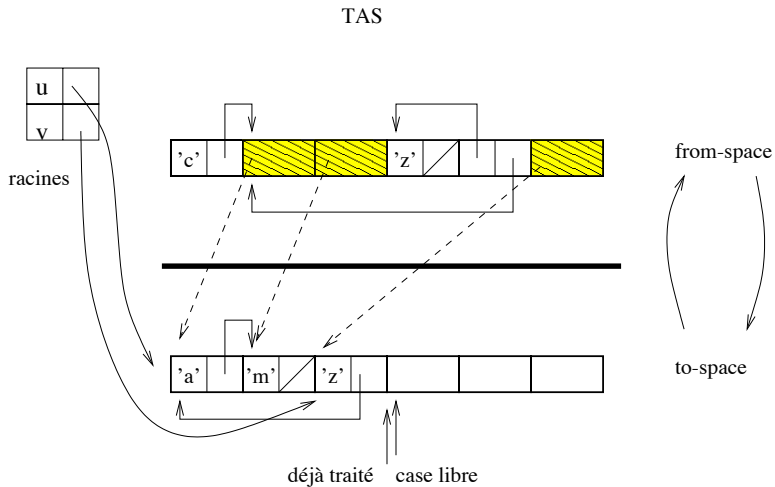
Récupération automatique de mémoire (GC)

Cours de Compilation Avancée (4I504)

Emmanuel Chailloux
Université Pierre et Marie Curie

Année 2015/2016 – Semaine 4

Modèle mémoire



Polymorphisme paramétrique: représentation uniforme

- ▶ 1 mot mémoire
 - ▶ valeurs immédiates (*int*, *char*, constructeurs constants)
 - ▶ autres valeurs : pointeur vers le tas (zone d'allocation dynamique)
- ▶ == teste l'égalité sur le mot (valeurs immédiate ou pointeur)

Allocation: explicite

Récupération: explicite ou implicite

Dangers de la récupération explicite

- ▶ pointeurs fantômes;
- ▶ zone mémoire inaccessible;
- ▶ extension de portée de variables locales (&).

⇒ difficulté de connaître la durée de vie d'une valeur!!!

Comparaison des récupérations

Intérêts: de la récupération implicite

- ▶ sûreté de la récupération
- ▶ propriétés sur le tas

Problèmes: liés (selon les techniques) à la récupération implicite

- ▶ mémoire non optimisée
- ▶ ralentissement de l'allocation, et de la récupération
- ▶ contraintes sur les techniques de compilation

Récupération automatique de mémoire

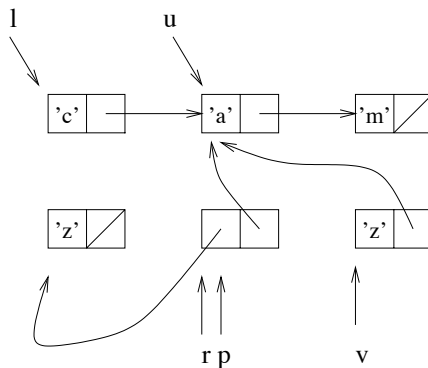
Quelques techniques de récupération automatique de mémoire

- ▶ compteurs de références
- ▶ algorithmes explorateurs
 - ▶ caractéristiques
 - ▶ Mark&Sweep
 - ▶ Stop&Copy
 - ▶ Mark&Compact
 - ▶ GC à générations
 - ▶ GC à racines ambiguës

Compteur de références

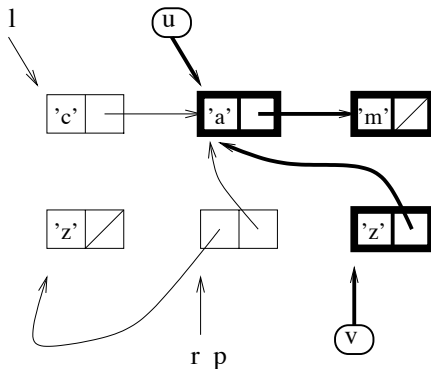
- ▶ associer à la zone mémoire allouée un compteur :
 - ▶ indiquant le nombre de pointeurs sur cet objet;
 - ▶ incrémenté à chaque partage de l'objet;
 - ▶ décrémenté quand un pointeur sur cet objet disparaît;
 - ▶ permettant de récupérer cet objet quand il vaut 0.
- ▶ avantage : libération immédiate de l'objet
- ▶ inconvénient : ne traite pas les objets circulaires.

Représentation mémoire (1)



```
1 let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
2 let v = let r = ( ['z'], u )
3   in match r with p -> (fst p) @ (snd p) ;;
```


Représentation mémoire (2)

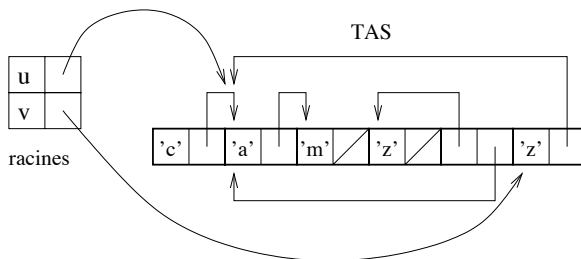


```
1 let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
2 let v = let r = ( ['z'] , u )
3   in match r with p -> (fst p) @ (snd p) ;;
```

Algorithmes explorateurs

- ▶ algorithmes explorateurs : techniques utilisables pour explorer à un instant donné l'ensemble des objets accessibles (graphe orienté).
- ▶ pointeur externe : un pointeur mémorisé dans un objet inaccessible au GC.
- ▶ racines de ce graphe :
un sous-ensemble des pointeurs externes à partir duquel on atteint tous les objets du graphe.

Représentation mémoire (3)



```
1 let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
2 let v = let r = ( ['z'] , u )
3           in match r with p -> (fst p) @ (snd p) ;;
```

Caractéristiques

sur la mémoire:

- ▶ facteur de récupération : quel % de la mémoire inutilisée est de nouveau disponible.
- ▶ compaction : toute la mémoire récupérée est-elle disponible en un seul bloc.
- ▶ localisation : les éléments d'un même objet sont-ils proches?

mesure de rapidité: temps d'allocation, de récupération

autres caractéristiques:

- ▶ GC conservatif
- ▶ datation des objets
- ▶ discrimination entre valeurs immédiates et pointeurs

Mark & Sweep : 2 étapes

- ▶ marquage des zones mémoires utiles à partir d'un *root set*.
- ▶ récupération des zones mémoires non marquées

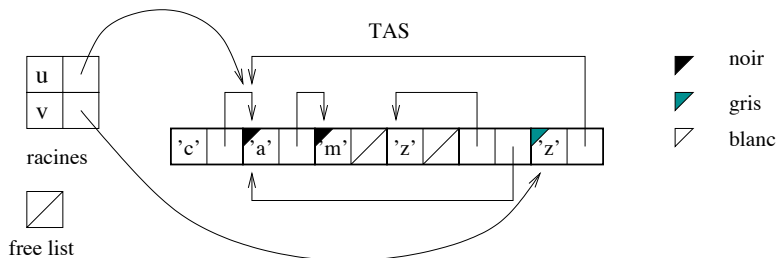
nécessite une information sur la taille des objets :

- ▶ indiquée à l'allocation
- ▶ ou calculée selon des pages mémoire d'objets de certaines tailles.

Mark & Sweep : exemple (1)

Mark, phase de marquage:

- ▶ gris : cellule marquée dont les fils ne le sont pas encore ;
- ▶ noir : cellule marquée dont les fils immédiats le sont aussi.



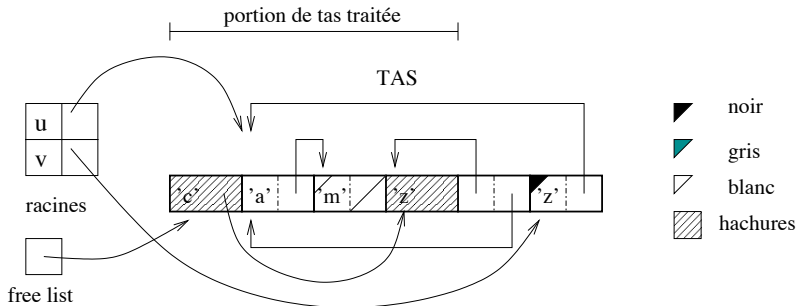
À la fin du marquage chaque cellule est blanche ou noire. Les cellules noires sont celles qui ont été atteintes depuis les racines.

Mark & Sweep : exemple (2)

Sweep, phase de récupération:

La récupération opère les modifications suivantes de coloration :

- ▶ noir devient blanc, la cellule vit ;
- ▶ blanc devient hachuré, la cellule est ajoutée à la *freelist*.



Mark & Sweep : caractéristiques

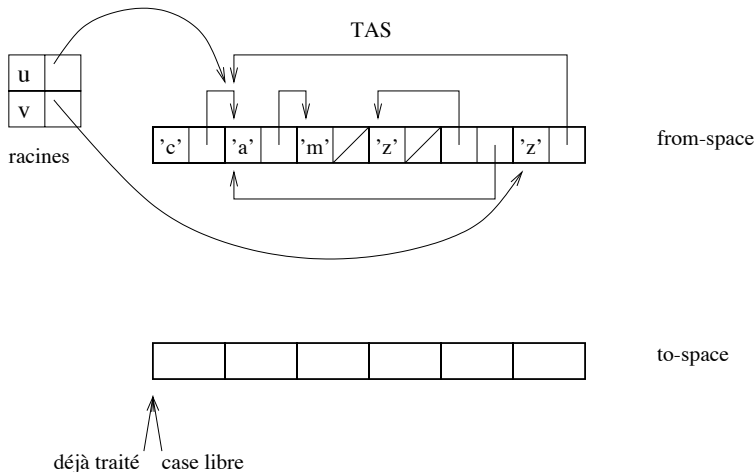
- ▶ dépend de la taille entière du tas (étape Sweep) ;
- ▶ récupère toute la mémoire disponible ;
- ▶ ne compacte pas la mémoire ;
- ▶ n'assure pas la localisation ;
- ▶ ne déplace pas les données .

Stop & Copy : 2 espaces

- ▶ utiliser une mémoire secondaire (to-space) pour recopier et compacter la mémoire à conserver (from-space).
- ▶ à partir du *set root* la partie utile de la zone from-space est recopiée dans l'espace to-space.
- ▶ En cas de partage, indiquer dans l'objet déplacé sa nouvelle adresse, qui sera retournée en cas de nouvelle copie.
- ▶ mise à jour du *set root*

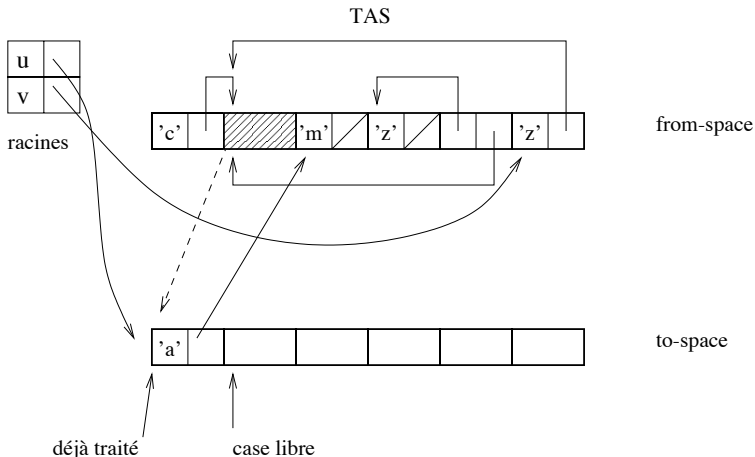
Stop & Copy : exemple (1)

2 espaces : `from_space` et `to_space`:



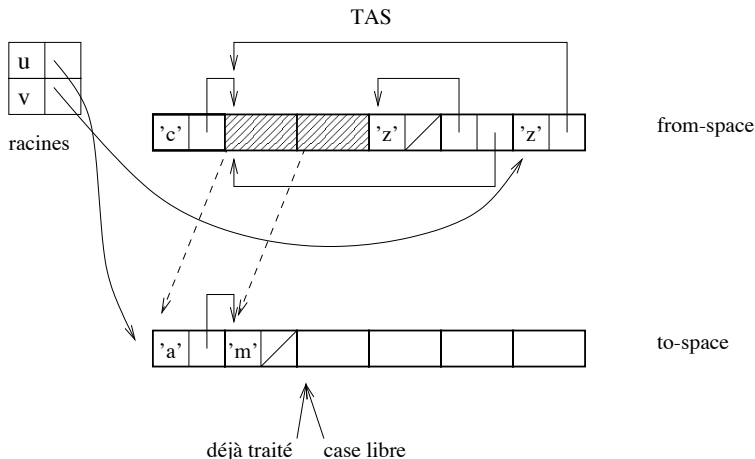
Stop & Copy : exemple (2)

À partir d'un ensemble de racines, on recopie la partie utile de la zone *from-space* dans l'espace *to-space*; la nouvelle adresse d'une valeur déplacée est conservée (le plus souvent dans son ancienne localisation) afin de mettre à jour toutes les autres valeurs pointant sur cette valeur.



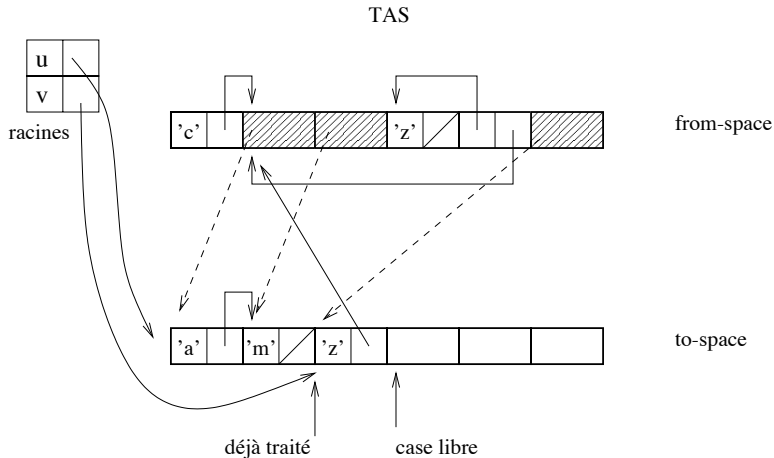
Stop & Copy : exemple (3)

Le contenu des cellules copiées forme les nouvelles racines. Tant que toutes ne sont pas traitées l'algorithme continue.



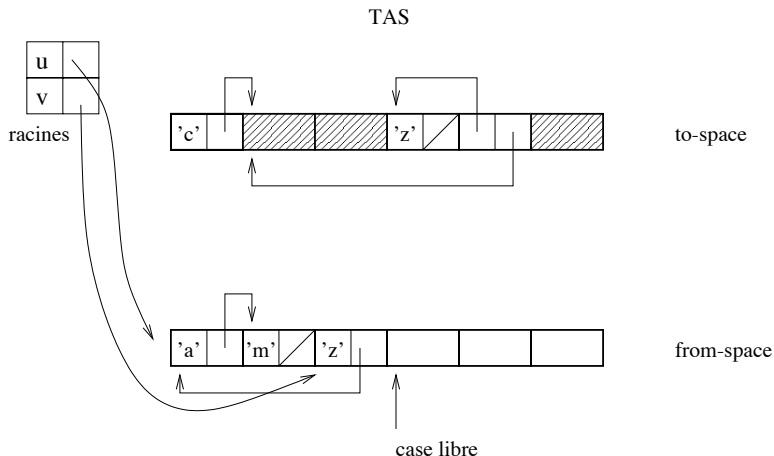
Stop & Copy : exemple (4)

En cas de partage, c'est-à-dire le déplacement d'une valeur déjà déplacée, on se contente de donner la nouvelle adresse.



Stop & Copy : exemple (5)

À la fin du GC, toutes les racines sont mises à jour pour pointer sur les nouvelles adresses. Enfin, les rôles des deux zones sont inversés en perspective du prochain GC.



Stop & Copy : caractéristiques

- ▶ dépend uniquement de la taille des objets à conserver;
- ▶ seule la moitié de la mémoire est disponible;
- ▶ compacte la mémoire;
- ▶ possibilité de localisation (parcours en largeur);
- ▶ n'utilise pas de mémoire supplémentaire (*from-space*+*to-space*);
- ▶ algorithme non récursif;
- ▶ déplace les données dans la nouvelle zone mémoire;

Mark & Compact : 2 étapes

- ▶ marquage des zones mémoires utiles à partir d'un *root set*.
- ▶ copie des données marquées en début de la zone mémoire (compactage)

nécessite une information sur la taille des objets :

- ▶ indiquée à l'allocation
- ▶ ou calculée selon des pages mémoire d'objets de certaines tailles.

Mark & Compact : algorithme

- ▶ **Mark, phase de marquage :**
 - ▶ idem au Mark & Sweep
- ▶ **Compact : 3 passes:**
 1. calcul des adresses
 2. mise à jour des pointeurs
 3. déplacement des objets

Mark & Compact : caractéristiques

- ▶ dépend de la taille des données vivantes du tas (étapes Mark & Compact) ;
- ▶ récupère toute la mémoire disponible ;
- ▶ compacte la mémoire ;
- ▶ possibilité de localisation (dépendante du parcours)
- ▶ déplace les données .

Autres GC : GC à générations

- ▶ modification du GC selon l'âge des objets
- ▶ GC rapide sur les objets jeunes
- ▶ GC rare sur les objets anciens
- ▶ technique utilisée en OCaml et en Java

Autres GC : GC à racines ambiguës

- ▶ ne pas marquer les objets (valeurs immédiates, pointeurs);
- ▶ adresses valides sont connues → discrimination;
- ▶ ne récupère pas forcément tous les objets morts;
- ▶ technique employée quand on compile vers C
- ▶ http://www.hpl.hp.com/personal/Hans_Boehm/gc/

à **générations**: 2 générations (ancienne et nouvelle)

- ▶ Stop&Copy sur la nouvelle (GC mineur)
- ▶ Mark&Sweep incémental sur l'ancienne génération (GC majeur)

caractéristiques:

- ▶ Un objet jeune qui survit à 1 GC change de zone.
- ▶ Conservation des pointeurs de zone ancienne vers zone jeune
- ▶ Si le Mark&Sweep échoue, alors un Stop&Copy un GC compactant est déclenché pour la génération ancienne.

Le module Gc permet de contrôler les paramètres du GC.

Module Gc (1)

- ▶ statistiques (type *stat* :
 - ▶ `Gc.stat : unit -> Gc.stat`
- ▶ contrôler les paramètres du tas (type *control*)
 - ▶ `Gc.get : unit -> Gc.control+`
`Gc.set : Gc.control -> unit`
- ▶ forcer le GC :
 - ▶ `Gc.minor() : unit -> unit`
 - ▶ `Gc.major() : unit -> unit`
 - ▶ `Gc.compact() : unit -> unit`

Module Gc (2)

Statistiques:

```
1  # Gc.stat();;
2  - : Gc.stat =
3  {Gc.minor_words = 112065.; promoted_words = 0.;
4   major_words = 60074.; minor_collections = 0;
5   major_collections = 0; heap_words = 126976;
6   heap_chunks = 1; live_words = 60074;
7   live_blocks = 11226; free_words = 66902;
8   free_blocks = 1; largest_free = 66902;
9   fragments = 0; compactions = 0;
10  top_heap_words = 126976; stack_size = 53}
```

et fonction d'affichage :

```
print_stat : out_channel -> unit
```

Module Gc (3)

```
1  # Gc.stat();;
2  - : Gc.stat =
3  {Gc.minor_words = 680793.; promoted_words = 7360.;
4   major_words = 117587.; minor_collections = 2;
5   major_collections = 4; heap_words = 126976;
6   heap_chunks = 1; live_words = 106362;
7   live_blocks = 23795; free_words = 20614;
8   free_blocks = 1; largest_free = 20614;
9   fragments = 0; compactions = 2;
10  top_heap_words = 126976; stack_size = 53}
11 # Gc.major();;
12 - : unit = ()
13 # Gc.stat();;
14 - : Gc.stat =
15 {Gc.minor_words = 702675.; promoted_words = 7360.;
16  major_words = 118290.; minor_collections = 2;
17  major_collections = 5; heap_words = 126976;
18  heap_chunks = 1; live_words = 106671;
19  live_blocks = 23880; free_words = 20305;
20  free_blocks = 18; largest_free = 19911;
21  fragments = 0; compactions = 2;
22  top_heap_words = 126976; stack_size = 53}
```


Module Gc (4)

Contrôle:

```
1 # let c = Gc.get () ;;
2 val c : Gc.control =
3   {Gc.minor_heap_size = 262144;
4     major_heap_increment = 126976;
5     space_overhead = 80;
6     verbose = 0;
7     max_overhead = 500;
8     stack_limit = 1048576;
9     allocation_policy = 0}
```

Module Gc (5)

Par exemple, le champ `verbose` peut prendre des valeurs de 0 à 127 activant 7 indicateurs différents.

```
1 # c.Gc.verbose <- 127 ;;
2 - : unit = ()
3 # Gc.set c ;;
4 - : unit = ()
5 # Gc.compact () ;;
```

affichage:

```
1 Heap compaction requested
2 <>Sweeping 9223372036854775807 words
3 Starting new major GC cycle
4 Marking 9223372036854775807 words
5 Subphase = 10
6 Sweeping 9223372036854775807 words
7 Compacting heap...
8 done.
```

Les différentes phases du GC sont indiquées ainsi que le nombre d'objets traités.

Gc en Java (1)

Allocation explicite (**new**) mais récupération automatique (GC).

- ▶ description des techniques du GC Java (Oracle)
 - ▶ générationnel, compactant, concurrent, parallèle :
 - ▶ génération jeune : création dans l'Eden, copie dans des zones de survie (objets datés) + 2 zones de survie : copie d'Eden vers S1, puis d'Eden+S0 vers S1 (objets d'âges différents)
 - ▶ génération ancienne : promotion d'objets ayant survécu à un certain nombre de GC mineurs
 - ▶ génération des permanents
 - ▶ concurrent pour la génération ancienne
 - ▶ parallèle : utilisation de plusieurs threads
- ▶ G1 (Java 1.7) : + incrémental pour éviter les longues pauses de récupération

Gc en Java (2)

- ▶ peut être déclenché explicitement

```
1 System.gc()
```

- ▶ et paramétré au lancement de la JVM
 - ▶ tailles initiales , maximales
 - ▶ choix du GC (séquentiel, parallèle, G1
 - ▶ trace : `-verbose:gc`
- ▶ possibilité de redéfinir une méthode `finalize` qui sera exécutée avant la libération d'un objet par le GC

Méthodes de finalisation

méthode `finalize()` héritée d'`Object` : permet une action sur un objet qui va être libéré par le GC :

- ▶ utile pour libérer une ressource système
- ▶ permet de gracier un objet
- ▶ déclenchement dépendant du GC
- ▶ en Java

```
1      protected finalize() throws Throwable{  
2          ...  
3      }
```

Pointeurs faibles

en OCaml et Java:

Un pointeur faible (*weak pointer*) est un pointeur dont la zone mémoire pointée est récupérable à tout moment par le GC.

Il peut être surprenant de parler d'une valeur qui peut disparaître à tout instant. En fait il faut voir ces pointeurs faibles comme un réservoir de valeurs encore disponibles. Cela s'avère particulièrement utile quand les ressources mémoire sont petites par rapport aux éléments à conserver. Le cas classique est la gestion d'un cache mémoire : une valeur peut être perdue, mais elle reste directement accessible tant qu'elle existe.

en Java

```
1  import java.lang.ref.*;
2  import java.util.*;
3
4  public class WeakReferenceTest{
5      public static void main(String[] argv){
6          ArrayList<Float> l = new ArrayList<Float>(5);
7          l.add(new Float(451.0));
8          l.add(new Float(37.2));
9
10         WeakReference<ArrayList<Float>> wr =
11             new WeakReference<ArrayList<Float>>(l);
12         l = null;
13
14         // ...
15
16         l = (ArrayList<Float>) wr.get();
17         if (l != null){
18             System.out.println(l.get(0));
19         }
20     }
21 }
```

en OCaml

pas de pointeurs faibles mais des tableaux de pointeurs faibles
module `Weak` définit le type abstrait `'a Weak.t` correspondant au type `'a option array`, vecteur de pointeurs faibles de type `'a`.

| fonction | type |
|---------------------|--|
| <code>create</code> | <code>int -> 'a t</code> |
| <code>set</code> | <code>'a t -> int -> 'a option -> unit</code> |
| <code>get</code> | <code>'a t -> int -> 'a option</code> |
| <code>check</code> | <code>'a t -> int -> bool</code> |

```
1 exception Found of int * Graphics.color array array ;;
2 let search_table filename table =
3   try
4     for i=0 to table.size-1 do
5       if i<>table.ind then
6         match Weak.get table.cache i with
7           Some (n,img) when n=filename -> raise (Found (i,img))
8           | _ -> ()
9       done ;
10      None
11 with Found (i,img) -> Some (i,img) ;;
```