Glaneurs de cellules

Gestion de la mémoire à l'exécution

- en C: `malloc, free`
- en Caml: *"toute valeur est éternelle"*
  - pas d'allocation explicite de mémoire
    (mais `ref` – et `new` – en font presque)
  - pas de désallocation explicite
- l'éternité ça prend de la place
  - certaines choses éternelles deviennent inutilisables
- gestion automatique de l'allocation dynamique

Grandes familles de GC

Exploration du tas

- les algorithmes les plus courants ont pour principe d'examiner le tas pour y détecter les zones libérables
- qui sont les objets vivants à l'instant t?
  - variables globales, variables locales, appels courants des fonctions → *ensemble des racines* (root set)
  - critère d'atteignabilité: les gens pointés par des gens vivants
  - approximation du caractère vivant: analyses à la compilation peuvent être plus précises
  - à ce sujet, il existe des méthodes d'analyse statique pour détecter qu'une donnée ne survit pas à la fonction qui l'a créée: allocation dans la pile

Marquer et balayer *(Mark & Sweep)*

- principe: deux phases
  - *reconnaître* à l'instant t l'ensemble des objets vivants
  - *libérer* les objets morts
- comment
  - à partir de l'ensemble des racines, marquer tous les objets atteignables (1 bit par objet en mémoire)
  - rassembler les objets non marqués dans une liste chaînée (cf. `malloc`) et démarquer les vivants
- problèmes:

  1 mémoire fragmentée — 2 coût proportionnel à la taille du tas
  3 localité mal gérée (cf. 1) — 4 une grande pause-GC
- algorithmes Mark-Compact
  - a priori encore plus lent: davantage de passes sur le tas

Comment reconnaître les pointeurs?

pointeur ou valeur immédiate?

- représentation des pointeurs en mémoire:
  - utilisation de quelques *bits d'étiquette*
  - en Caml: valeur sur le tas ou immédiate (`int`: 31 bits)
- selon les langages et les implémentations, plus ou moins d'information sur le type des objets en mémoire
  - contient ou non des pointeurs vers d'autres objets
  - si le langage est fortement typé, possibilité de ne pas étiqueter
    (l'ensemble des racines suffit, puis on "suit les types")
- approximation (algorithmes *conservatifs*) pas d'étiquette, on regarde si adresse valide, *alignée*

  ouvre la voie au GC pour langages sans GC (p.ex. C, en demandant au programmeur d'être gentil)

Stop & Copy

- principe:
  - le tas est découpé en deux zones de même taille
  - déplacer les objets vivants d'une zone du tas vers une autre
  - quand une passe est finie, la zone originelle est libre
  - et la zone pour l'allocation est contigüe
- exemple: algorithme de Cheney
  - exploration des objets vivants (dessin)
  - objets pointés plusieurs fois: si pointeur vers *fromspace*, on regarde s'il y a un *forwarding pointer*
- discussion:
  - on n'examine qu'une fois chaque objet vivant
  - coût proportionnel à la quantité d'objets vivants à l'instant t
  - pas de fragmentation, localité assez bien gérée
  - on coupe la mémoire en deux (*fromspace, tospace*)

Compter les références - principe

- approche sensiblement différente:

  maintenir pour chaque entité allouée dans le tas un compteur indiquant combien d'autres entités pointent vers elle

```
x = "hip";   "hip" 1              allocation
y = x;       "hip" 2              incrémentation
y = "hop";   "hip" 1  "hop" 1     incrémentation /
                                    décrémentation
```

- quand un compteur arrive à 0, libérer l'espace mémoire correspondant
- propagation de la libération

  un objet qui est libéré peut avoir des pointeurs vers d'autres objets: décrémenter → libération récursive
- difficulté: structures cycliques

  - p.ex. structure chaînée: `li = 3::2::5::2::li`, chaque maillon a un compteur à 1

Compter les références - observations

- discussion
  - mémoire gérée "en temps réel": approche *incrémentale*
    - on peut aussi retarder la propagation pour "assurer des délais"
  - mais gourmandise en mémoire
    - typiquement 1 mot mémoire pour l'indice
    - parfois on met une limite (limite atteinte ⇒ objet non effaçable directement)
  - ce temps de nettoyage équiréparti peut être long
    - ex.: variables ayant une courte durée de vie sur la pile, traitement inutile → variantes du comptage de réfs
- également: variantes distribuées
  - questions de synchronisation

Organisation

- cours de prog le **jeudi 4 janvier** de 14h30 à 17h30
- le sujet d'examen de l'an dernier est disponible sur la page du cours
  - sont autorisés: transparents du cours, sujets/corriges de TP

mais aucun ordinateur

Approches incrémentales

Ajouter l'incrémentalité

- comptage de références: le GC se fait petit à petit, mais est coûteux
- algorithmes M&S, S&C: GC = opération "atomique" *(!)*
- idée: mener progressivement la phase de reconnaissance des objets vivants (dans un *Mark & Sweep* ou un *Stop & Copy*)

phase commune aux deux grands types d'approche

Parallélisme avec mémoire partagée

- point de vue: la mémoire est un ensemble de données sur lequel travaillent *en parallèle* deux agents:
  - *mutator* le programme
  - *collector* le GC

  l'enjeu est de préserver une version conservative de la mémoire (ne pas jeter à la poubelle des choses utiles)
- le scénario varie suivant l'approche
  - M&S: *single writer, multiple readers*

    un écrivain, un lecteur, en fait
  - S&C: *multiple writers, multiple readers*

    deux écrivains *(les deux peuvent modifier un champ)*

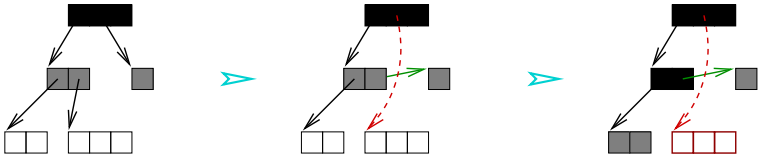
Algorithmes incrémentaux

- on veut représenter pour le raisonnement la vision qu'a *collector* de la mémoire à un instant donné

  on introduit un marquage tricolore **conceptuel** des objets

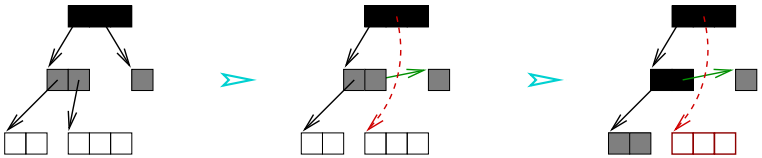
  blanc pas encore visité gris visité mais pas ses fils

  noir visité, ainsi que ses fils
- objets marqués en gris (= "crête de la vague")
  - Cheney = entre les pointeurs
  - M&S = sur la pile (ou file)
- invariant à garantir: toujours un gris pour aller d'un noir à un blanc



Préserver l'invariant: barrières

→ algorithmes introduisant des formes de synchronisation *(barrières)* entre *mutator* et *collector*



- barrière en lecture (S&C): interdire à *mutator* de "voir" un objet blanc (cf. + loin)
- barrière en écriture (M&S): problème lorsque *mutator*
  - 1 installe un pointeur vers blanc dans un noir, **et**
  - 2 détruit le pointeur préexistant
  - solutions proposées:

    éviter que l'une des conditions soit vérifiée

Barrières en écriture

- *"snapshot at beginning":*
  - lorsque l'on réécrit un pointeur, on stocke la valeur écrasée pour la traiter par la suite

    (l'invariant peut d'ailleurs du coup être violé)
  - tous les objets alloués sont noirs

    → on nettoie suivant *(l'état du début + les nouveaux objets)*
- autre approche (Dijkstra et al.):
  - lorsque l'on écrit un pointeur dans un objet noir, colorier ce dernier en gris
  - les objets alloués sont blancs (on parie sur une vie courte)

    politique plus agressive
  - il faut potentiellement ré-examiner certains objets

Barrières en lecture – l'algorithme de Baker

- variante incrémentale de l'algorithme de Cheney
  barrière en lecture: pas le droit de regarder un objet blanc
- au début, copie dans *tospace* les objets directement atteignables
- on accompagne mutator: il ne doit jamais regarder dans *fromspace* (copie imposée)
- version *sans copie*: le boulanger devient meunier: dessin
  - 4 zones: Free (pour l'allocation), From (objets inspectés), To et New (vides au début du GC)
  - à la fin: pas de gris dans To, ce qui reste dans From est glané, le nouveau From est égal à To + New le moulin a tourné
- on fait au passage des *approximations conservatives*:
  - allocations en noir
  - objets peuvent être recopiés puis devenir inactifs

Combiner les approches: glaneurs générationnels

Glaneurs générationnels

- idée: coupler plusieurs GCs, "nettoyages" de moins en moins fréquents
- objets jeunes: durée de vie courte, GC rapide
- objets anciens: ont survécu à quelques GCs, GC lent réordonnant la mémoire
- indépendance GC jeune/GC vieux?
  - fonctionnel pur: seulement pointeurs "jeunes vers anciens"
    - GC jeune "indépendant" *(Directed Acyclic Graph)*
  - Caml: on stocke les pointeurs des anciens vers les jeunes
    - fonctionnel pur avec évaluation retardée: idem

Le GC de OCaml

- deux générations:
  - GC mineur (jeunes): *Stop and Copy*
  - GC majeur (anciens): *Mark and Sweep* incrémental + algorithme de compaction
- *tospace* du GC mineur: zone du GC majeur
- allocation
  - dans le jeune: linéairement
  - dans le vieux: liste cyclique

OCaml – GC majeur

- à chaque GC mineur, un peu de GC majeur est fait, en fonction de la taille d'objets vivants issus du GC mineur
  (heuristique pour limiter la durée des GCs majeurs)
- un GC majeur:
  - finir la phase courante de M&S
  - éventuellement compactage de la mémoire
- couleur des objets nouvellement alloués
  - blanc si on est en phase de Sweep, dans une zone déjà balayée
  - noir sinon

GC de OCaml – structures de données

- zones en mémoire pour la jeune et vieille génération
- comme indiqué précédemment:

```
# let ancien = ref [1] ;;
(* ...  du temps ...  *)
# let jeune = [2;5;8] in ancien := jeune ;;
```

  table de références anciens→ jeunes, effacée à chaque GC mineur
- une pile de pointeurs sur les objets gris
- d'autres choses encore (gestion des pages mémoire)
- écrire un vrai GC correct, c'est *très compliqué*
- celui de Caml a été prouvé correct sur machine

(Doligez, Gonthier)

Libraire Gc de Caml

- une librairie pour avoir des informations sur le fonctionnement et l'état du GC
  - taille des différentes zones
  - nombre de glanages effectués (mineur et majeur)

    DÉMO `run_gc.ml`
- associer une fonction
  - au déclenchement d'un GC majeur
  - au moment (≈) où une valeur donnée est obsolète

    DÉMO `finalise.ml`