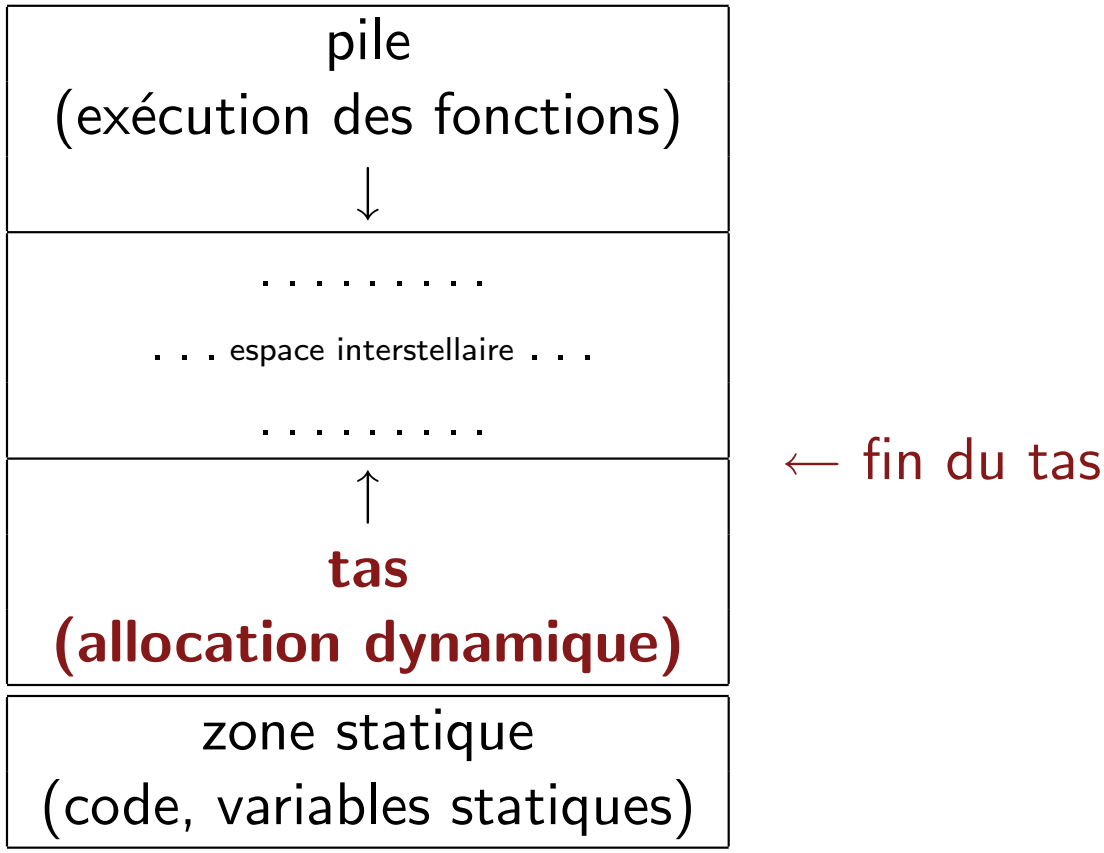
Mémoire

Découpage de la mémoire



Pile et tas

- `let f x y = ...` `let g a = h(f a (a+1))`
  - lorsqu'on exécute `f`, `x` et `y` ont un sens, pas `a`
  - la pile implémente à la fois le flot du calcul et la portée
- on empile des enregistrements d'activation

  ...dont le contenu est plus clair en C qu'en Caml

  cf. `int f(int a, struct machin c){ int i,j; double d; ...}`
- `let crée_init k = let r = ref [k;k;k] in r`
  - si `[k;k;k]` est stocké dans la pile
    - soit on va écraser après être sortis de `crée_init`
    - soit on le recopie à chaque fois

      et quid si la taille n'est pas prévisible?
  - on a une donnée
    - pour laquelle on *alloue de la mémoire* en cours d'exécution (ici, à chaque appel à `crée_init`)
    - qui doit "survivre" à l'appel de fonction qui l'a créée
- le tas est la zone de l'*allocation dynamique*

  le code qui tourne pour l'allocation mémoire n'a pas été préparé lors de la compilation, comme sur la pile

Allocation dynamique de la mémoire

- en C
  - variables *statiques* (durée de vie = celle du programme), *automatiques* (un bloc, typiquement corps d'une fonction), *dynamiques* (tout est possible)
  - allocation dynamique *explicite*
    - pour des tableaux, des structures ...
    - C++, Java: `new` (création d'un objet) alloue dans le tas
  - les tableaux sont toujours passés par adresse, mais peuvent être alloués dans la pile

    ```
    void f(int a){ float tab[10];...}
    ```
- en Caml
  - la distinction *boxed/immédiat* est en amont
    - dès que c'est boxed, c'est dans le tas

      ```
      let f x = let li = 1::x in List.length li
      ```
  - Caml gère le tas tout seul

    (le `li` ci-dessus serait probablement une variable locale dans un monde à la C)

Allocation dynamique en C

Allocation dynamique: la fonction `malloc`

- principe: `malloc` réserve une zone de taille donnée
- `void * malloc (size_t SIZE)`
  - p.ex. `(struct machin *) malloc(12)`: *cast explicite*
  - renvoie `NULL` s'il n'y a plus de place pour allouer
  - la zone allouée n'est **pas** initialisée ($\neq$ `brk`: cf. + loin)
  - elle est alignée, contigüe
- DÉMO `adresses.c, ecrase.c`

Désallouer: `free`

- une zone allouée par `malloc` est réservée jusqu'à la fin du programme...
- ...à moins de la libérer explicitement: `free`

  `void free (void *PTR)`
- libérer peut altérer: ne pas aller voir après avoir libéré
- libérer un bloc *comme il a été alloué*

  il est plus sage de ne pas toucher au pointeur entretemps
- en pratique, pas besoin de libérer à la fin du programme

  `info libc:` *There is no point in freeing blocks at the end of a program, because all of the program's space is given back to the system when the process terminates.*
- en théorie, c'est plus sain
- DÉMO `test_free[0,1,2].c`, `pointNULL[1,2].c`

malloc et free – exemples

```
struct chain
{
  struct chain *next;
  char *name;
}

struct chain *cons(struct chain *current_chain, char *x){
  struct chain *ptr
    = (struct chain *) malloc (sizeof (struct chain));
  ptr->name = x;
  ptr->next = current_chain;
  return ptr;
}

void free_chain (struct chain *chain)
{
  while (chain)    /* chain!=0 */
  {
    struct chain *next = chain->next;
    free (chain->name);
    free (chain);
    chain = next;
  }
}
```

Autres fonctions pour la gestion dynamique de la mémoire

- `realloc`

  `void * realloc (void *PTR, size_t NEWSIZE)`

  - peut faire intervenir une recopie
  - si taille plus petite, c'est le début qui est gardé

```
float *tab = NULL;
int size = 0;                          /* size est statique */

void assign (int i, float x){
  if (i<size){ tab[i] = x; return; }
  { size=i; tab = (float *) realloc(tab, size*sizeof(*tab))};
  if (tab == NULL) { ... }
  tab[i] = x;
}
```

- allouer de très grandes zones: `mmap` *cf ASR*

Autres fonctions, suite

- `void * memcpy (void *restrict TO, const void *restrict FROM, size_t SIZE)`
  - recopie `SIZE` octets de `FROM` vers `TO`
  - `const`, `restrict`: qualificatifs de types
    - `const`: on n'écrit pas dans cette zone
    - `restrict`: spécifie que `memcpy` est codée en faisant l'hypothèse qu'on n'a pas d'aliasing entre `TO` et `FROM`
      comportement non spécifié si `FROM` et `TO` se marchent dessus
- `#include <memory.h>`

```
char * savestring (char *ptr, size_t len)
{
  char *value = (char *) malloc (len + 1);
  value[len] = '\0';
  return (char *) memcpy (value, ptr, len);
}
```

Fonctionnement de `malloc`/`free`: principes

Allocation/désallocation dynamique

- au cours de son exécution, un programme peut allouer et désallouer une grande quantité de données
  - ceci de manière a priori non disciplinée (pas de régime FIFO, par exemple)
  - résultat: le tas se présente comme un "damier" très irrégulier
    fragmentation
- `malloc` et `free` fournissent une interface (abstraction) pour gérer l'interaction avec cette zone de la mémoire
  - réutiliser les zones disponibles
  - garder une taille raisonnable pour le tas
    - enjeux en espace
    - mais aussi en temps
      on ne veut pas passer son temps à allouer/désallouer
  - favoriser si possible la *localité*
    - rapprocher les zones qui sont accédées "ensemble"
- et si on n'a plus de place dans le tas?
  interaction avec le système: `brk`, `sbrk`

Le point de rupture

- plus petite adresse non utilisée du tas: *breakpoint*
  - accès à une adresse au-dessus du breakpoint (et pas dans la pile): absurde, ou *signal* (et ça plante)
- primitive `brk` `int brk (const void*)`
  - argument: nouvelle adresse de fin du tas
  - renvoie 0 ou -1 (-1: échec – valeur sous l'espace des données, trop grande, etc.)
  - lorsque le tas grandit, la nouvelle zone est initialisée à 0
- primitive "de plus haut niveau": `sbrk`

  `void* sbrk(int)`
  - on donne un incrément (pouvant être < 0), on obtient l'adresse du point de rupture
    - si on passe 0, on récupère l'adresse de fin du tas
- NB: si on joue avec `brk`, on doit renoncer à `malloc` (et donc à de nombreuses librairies), à cause d'interférences possibles

Organisation du tas

- les blocs dans le tas:

...	réservé	réservé	libre	réservé	libre	réservé	...

- chaque bloc est
  - aligné en mémoire (accès aisé)
  - précédé par un *en-tête*

en-tête	..... *bloc* .....

    - l'en-tête contient en particulier la taille du bloc
- première approche:

  `malloc` et `free` tiennent à jour les en-têtes et une liste chaînée des blocs libres
- les grandes idées, sur une implantation simplifiée

  (tirée du livre de Kernighan et Richie)

  DÉMO `kr_simplifie.c`

Algorithme de Doug Lea – principes

- forme d'un bloc:

`statut`	`taille`	*données*	`taille`

  - statut: libre / occupé
  - taille à la fin: pouvoir être parcouru dans les deux sens
- gestion des blocs libres: "par calibre", 128 tailles
  - 16, 24, 32,. . . , 512: taille exacte
  - 576. . . 2^{31}: triés par taille croissante dans chaque intervalle
- idées de base:
  - `malloc`: on choisit le meilleur bloc (*best fit* et non plus *first fit*)
  - `free`: on fusionne les blocs adjacents
- heuristiques pour favoriser la localité

  (même moment d'allocation → même vie)

  1. bloc de bonne taille / 2. zone coupée récemment / 3. best fit

  faire un (dés)allocateur **générique**, c'est compliqué
- retrouver certaines idées: DÉMO `en_tete.c`

Raffinements ultérieurs

- le dernier bloc *(wilderness chunk)* est considéré comme de taille grosse, + grosse que tous les autres *(car on peut utiliser `brk`)*
- heuristiques autour de la taille des blocs alloués
  - certaines applications font beaucoup d'allocations/libérations de petits objets, de taille constante (p.ex. nœuds d'un arbre)
    - assouplir la politique de fusionnement
    - "précuire" une zone dédiée avec des blocs de la bonne taille
  - au contraire, un développement à objets avec beaucoup de classes voit la taille des blocs alloués varier considérablement
- il peut à l'occasion être intéressant de programmer sa propre allocation dynamique sur certains types de données pour des questions d'efficacité *(p.ex. en C++: surcharge de `new`)*

Allocation dynamique: difficultés

- l'infrastructure pour l'allocation dynamique (en-têtes) est accessible: on peut tout casser
  - p.ex. sortir du bloc et écraser l'en-tête du bloc suivant
- formes d'échec:
  - `‘‘malloc: corrupt arena’’`
  - ça plante (`core dumped`)
  - ça ne plante pas (pire)
- libération: de nombreuses situations incohérentes potentielles
  - libérer au mauvais endroit (au milieu d'un bloc, sur la pile)
  - deux fois `free` du même pointeur `p`
    - si la zone a été réallouée (pour `q`), `q` devient un *pointeur fantôme* (+ erreur difficile à détecter)
    - pire: la zone a été fusionnée avec ses voisines
  - deux pointeurs sur la même zone, et un `free`
  - zone pas libre, pas référencée: *fuite de mémoire*
- bonne pratique: `free(p); p = NULL;` *(macros)*

Garde-fous

approches envisagées pour lutter contre les bugs dans la gestion de la mémoire

- allouer “un peu plus” pour chaque bloc, afin de limiter la casse si on écrit trop loin
- maintenir des tables indiquant l'organisation et le statut des blocs
  - l'infrastructure est à part
  - un outil comme `purify` *(“debugger de mémoire”)* maintient une image de la mémoire et détecte les accès suspects
    - écriture après libération
    - lecture avant qu'une écriture ait eu lieu

Allocation dynamique: approches

- les primitives à la `malloc/free` permettent une gestion explicite de l'allocation dynamique
  - c'est source d'erreurs
  - mais on sait ce qu'on fait
- gestion automatique de l'allocation dynamique:

  glaneurs de cellules

  (garbage collector)

  - à l'origine, pour LISP (McCarthy)
  - de nombreux langages de programmation fonctionnelle
  - Java