Récursion terminale; gestion de la mémoire en OCaml

Crédits

- Ce site develooppez.com

Un exemple ♡

Considérons le programme suivant. Il calcule la somme des entiers de 0 à 1 000 000 :

```
let rec somme n =
  if n=0 then 0 else n + somme (n-1)
let v = somme 1_000_000
```

Compilation, exécution :

```
$ ocamlopt somme.ml -o somme
$ ./somme
Fatal error: exception Stack_overflow
```

Il y a débordement de pile : le nombre de stack frame est trop grand.

Analyse ♡

- On empile puis dépile les stack frame

Analyse ♡

- On empile puis dépile les stack frame
- Chaque stack frame contient une sauvegarde des registres du processeur ; un espace pour stocker la valeur de retour ; le paramètre ; l'adresse de retour

Analyse ♡

- On empile puis dépile les stack frame
- Chaque stack frame contient une sauvegarde des registres du processeur ; un espace pour stocker la valeur de retour ; le paramètre ; l'adresse de retour
- Entrons la commande suivante

```
$ ulimit -s
8192
```

On obtient donc que la taille de la pile d'appel est de 8 Mo.

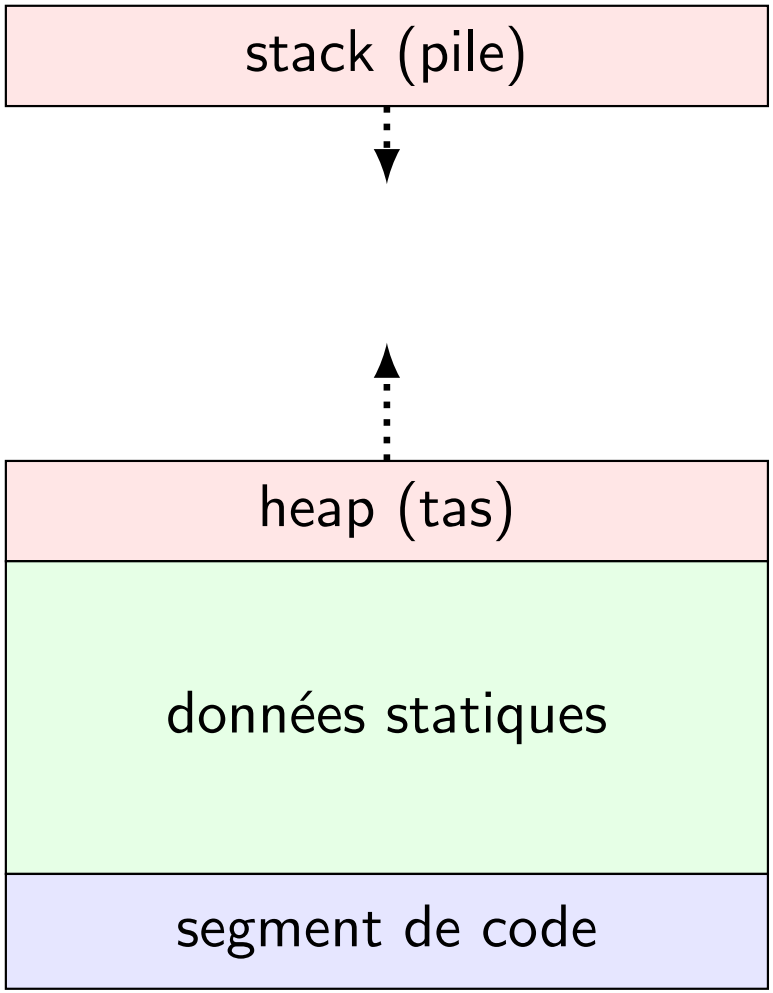
Analyse ♡

- On empile puis dépile les stack frame
- Chaque stack frame contient une sauvegarde des registres du processeur ; un espace pour stocker la valeur de retour ; le paramètre ; l'adresse de retour
- Entrons la commande suivante

```
$ ulimit -s
8192
```

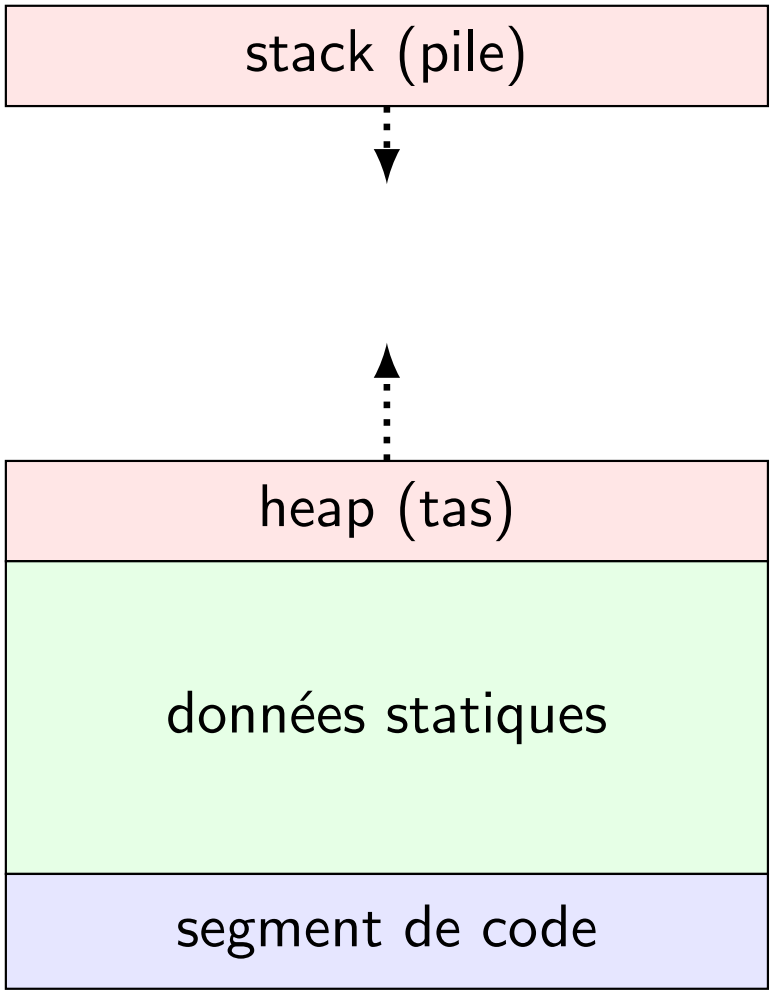
  On obtient donc que la taille de la pile d'appel est de 8 Mo.
- Avec `somme 1_000_000` on empile donc $1\,000\,000 + 1$ stack frame d'au moins 8 bytes. On comprend que la taille allouée à la pile soit dépassée.

Organisation de la mémoire (shéma simplifié)♡



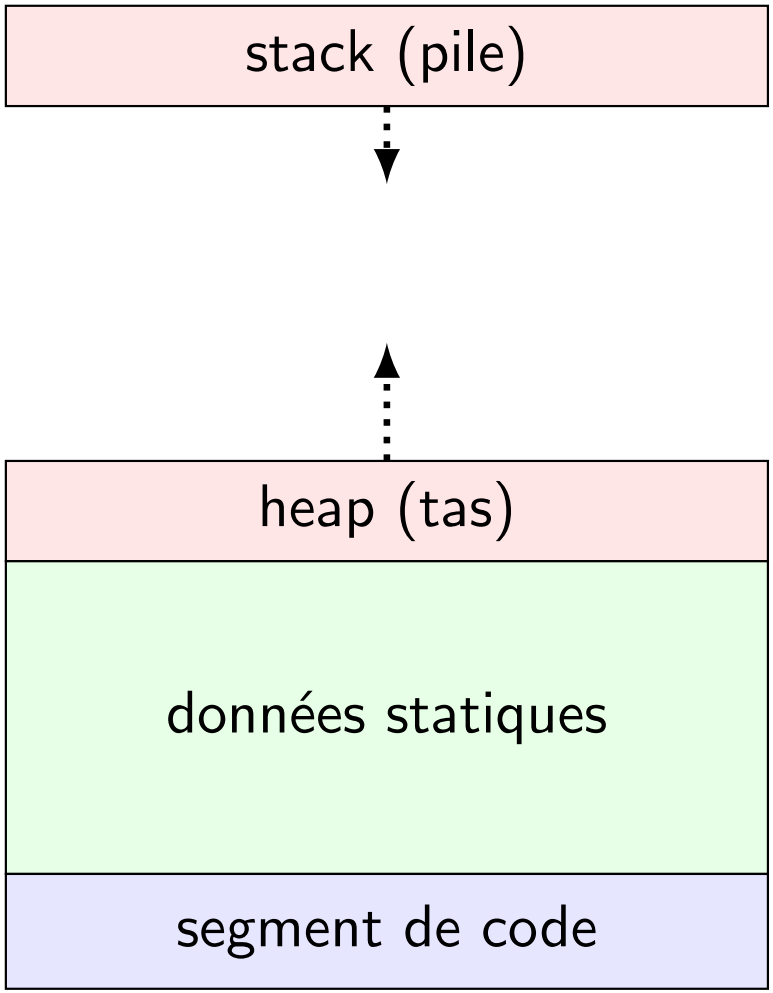
- La zone de données statiques contient les constantes présentes dans le code source du programme comme les constantes chaînes.

Organisation de la mémoire (shéma simplifié)♡



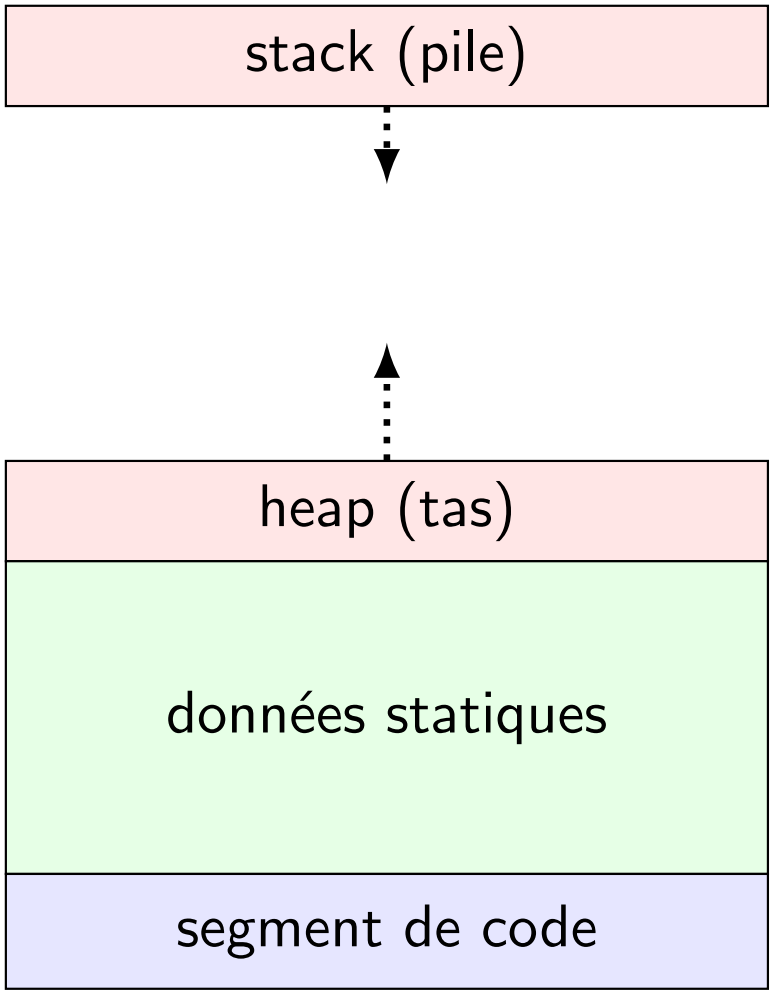
- La zone de données statiques contient les constantes présentes dans le code source du programme comme les constantes chaînes.
- Toutes les valeurs en OCaml sont des pointeurs sur des données dans le tas. Les var. locales de la pile ne peuvent être que d'une des 4 catégories suivantes :

Organisation de la mémoire (shéma simplifié)♡



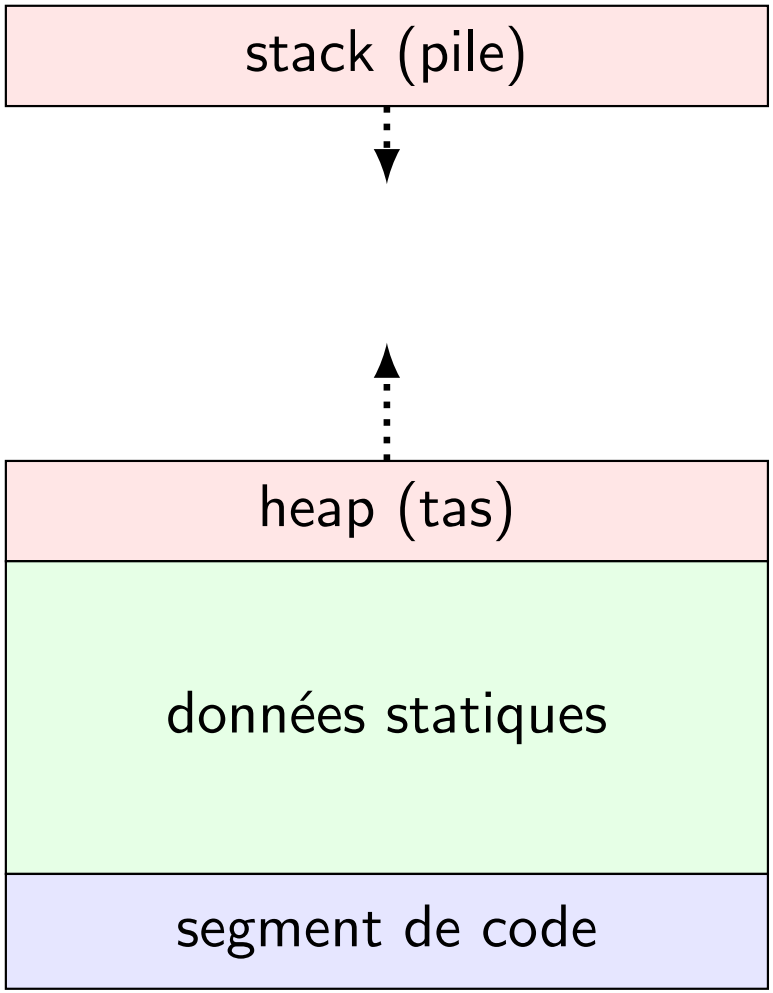
- La zone de données statiques contient les constantes présentes dans le code source du programme comme les constantes chaînes.
- Toutes les valeurs en OCaml sont des pointeurs sur des données dans le tas. Les var. locales de la pile ne peuvent être que d'une des 4 catégories suivantes :
  - les entiers `int`, les caractères `char`

Organisation de la mémoire (shéma simplifié)♡



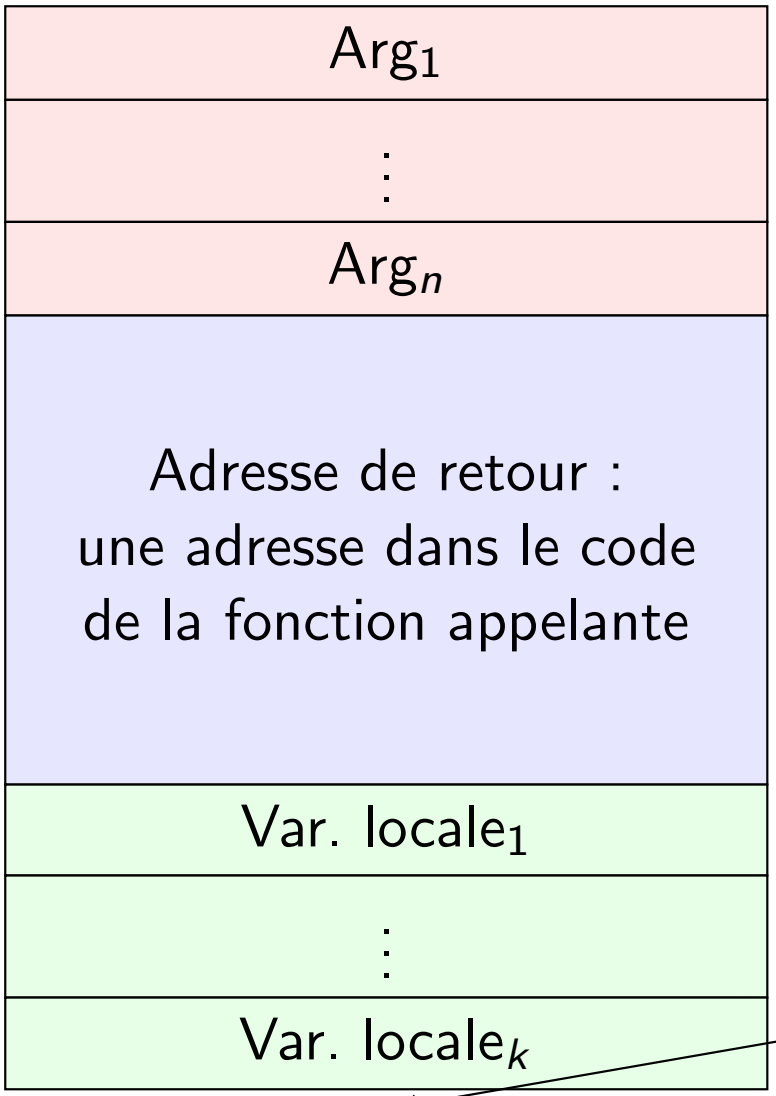
- La zone de données statiques contient les constantes présentes dans le code source du programme comme les constantes chaînes.
- Toutes les valeurs en OCaml sont des pointeurs sur des données dans le tas. Les var. locales de la pile ne peuvent être que d'une des 4 catégories suivantes :
  - les entiers `int`, les caractères `char`
  - le type `unit`

Organisation de la mémoire (shéma simplifié)♡



- La zone de données statiques contient les constantes présentes dans le code source du programme comme les constantes chaînes.
- Toutes les valeurs en OCaml sont des pointeurs sur des données dans le tas. Les var. locales de la pile ne peuvent être que d'une des 4 catégories suivantes :
  - les entiers `int`, les caractères `char`
  - le type `unit`
  - les constructeurs sans arguments dans les types sommes (ils sont représentés en interne par des entiers)

Schéma général d'une stack frame

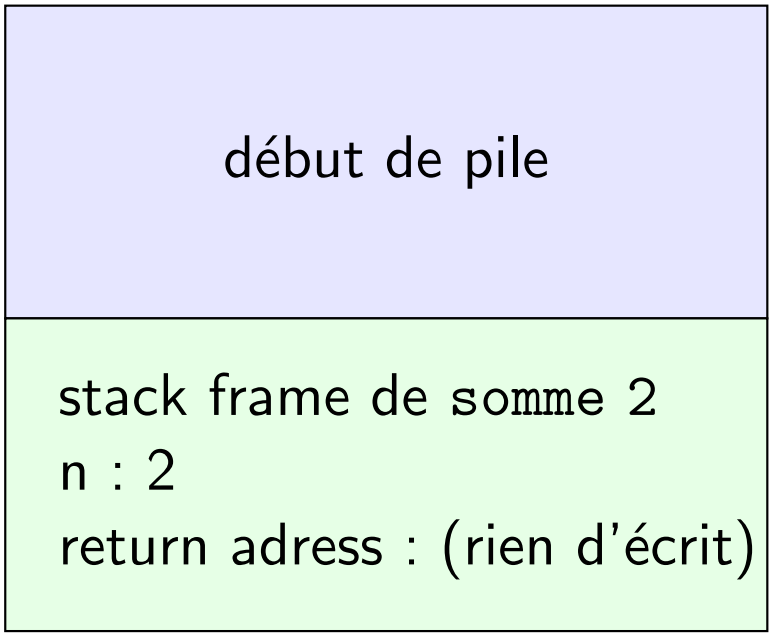


Adresse début stack frame
registre EBP

La stack frame courante
se situe entre les adresses
pointées par EBP et ESP

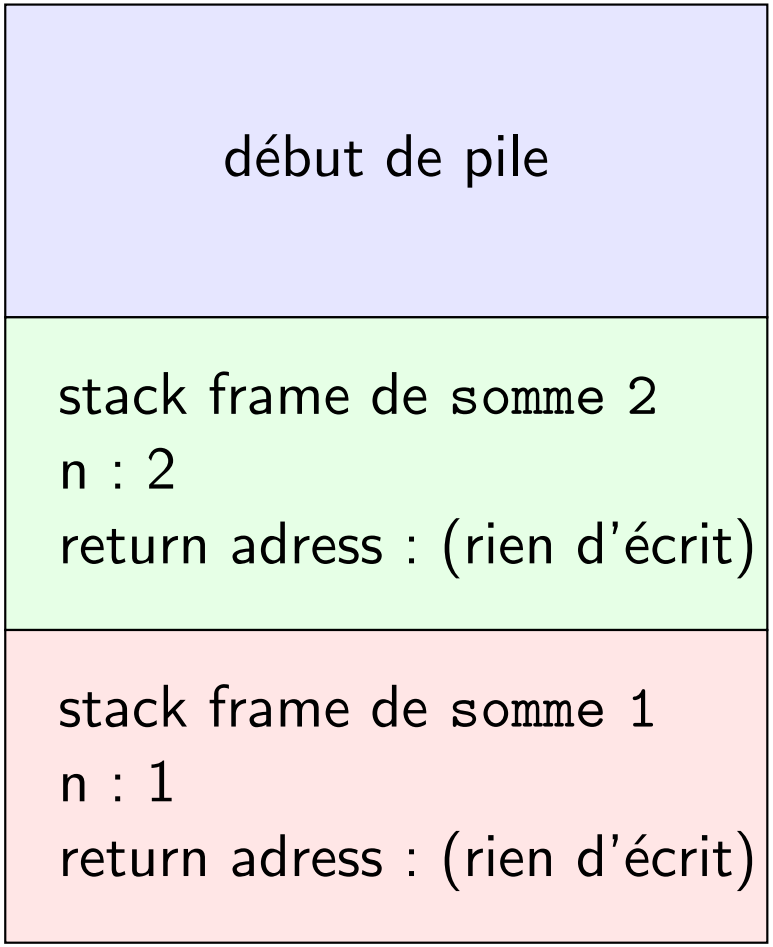
Adresse du
sommet de pile d'exécution
registre ESP

Schéma simplifié de l'appel de `somme 2`



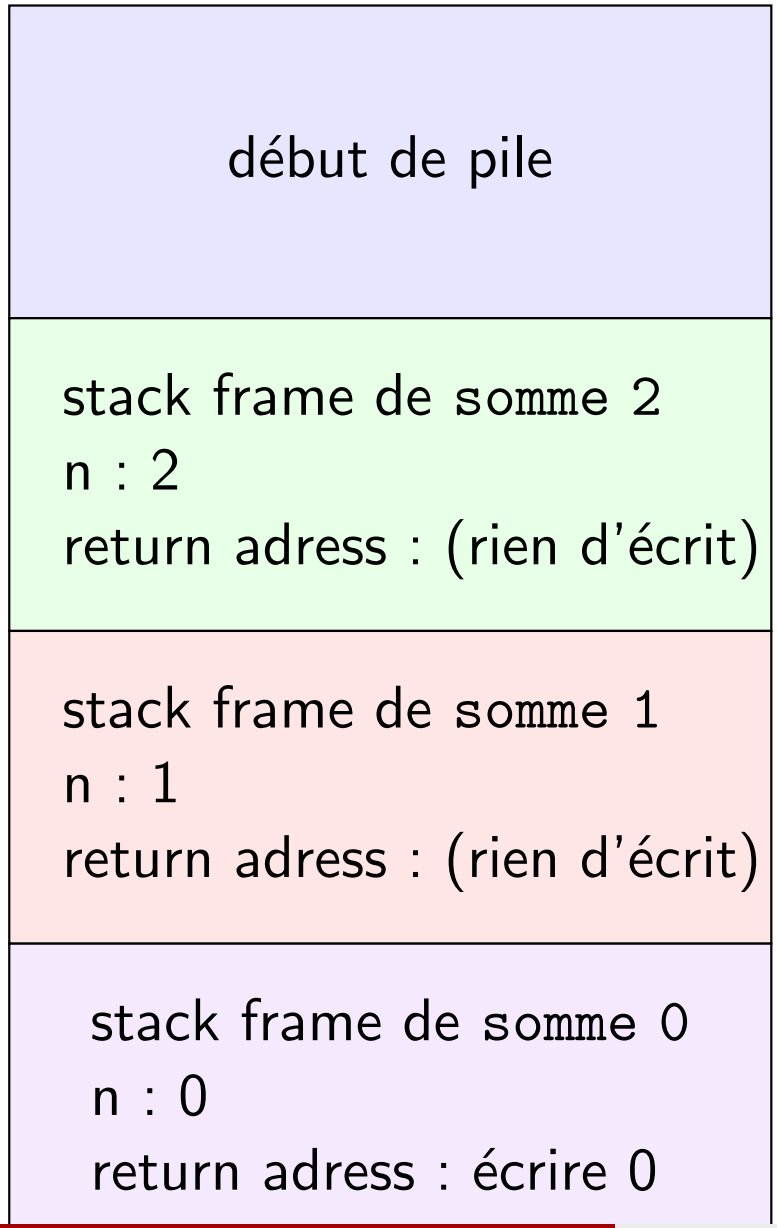
Mémoire occupée :
1 entier ; 1 adresse

Schéma simplifié de l'appel de `somme 2`



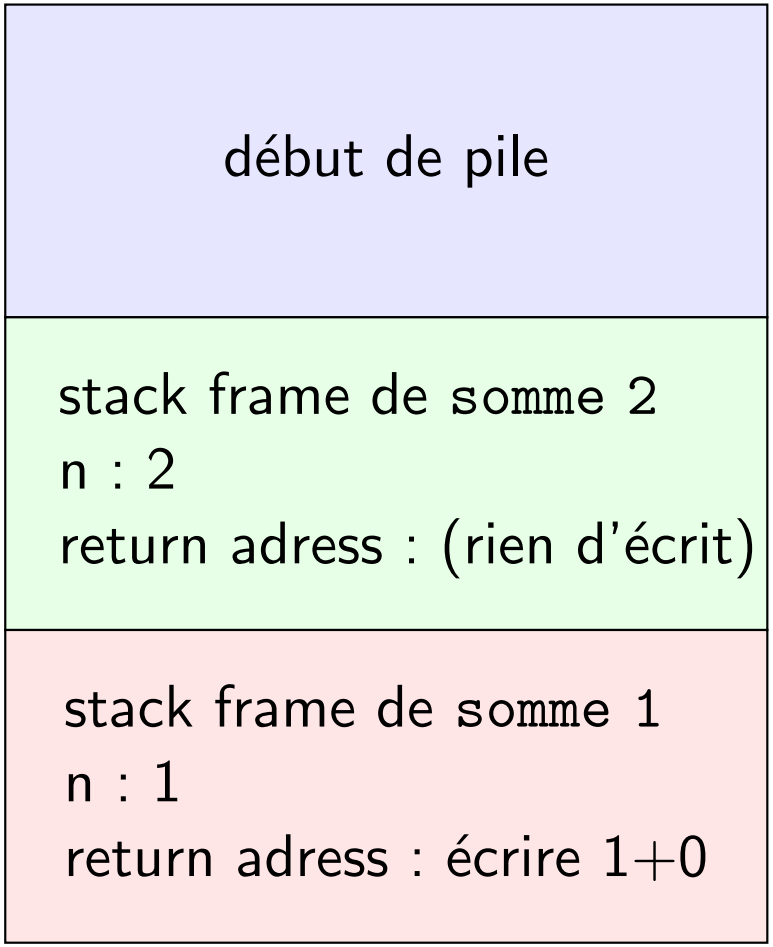
Mémoire occupée :
2 entiers ; 2 adresses

Schéma simplifié de l'appel de `somme 2`



Mémoire occupée :
3 entiers ; 3 adresses

Schéma simplifié de l'appel de `somme 2`



Mémoire occupée :
2 entiers ; 2 adresses

Schéma simplifié de l'appel de `somme 2`

Mémoire occupée :
1 entier ; 1 adresse

Ramasse miette

- En OCaml, la libération des zones mémoires est faite automatiquement par un algorithme appelé *ramasse miette* (en anglais Garbage Collector -GC-).

Ramasse miette

- En OCaml, la libération des zones mémoires est faite automatiquement par un algorithme appelé *ramasse miette* (en anglais Garbage Collector -GC-).
- Le GC agit pendant l'exécution du programme. Il détermine quelles zones mémoires dans le tas sont devenues inutiles. Il les libère automatiquement.

Gestion de la mémoire par l'OS

- L'OS gère l'espace occupé par les segments durant l'exécution du programme. C'est lui qui assure l'intégrité de ces différents segments et qui empêche par exemple que la pile écrive dans le tas.

Gestion de la mémoire par l'OS

- L'OS gère l'espace occupé par les segments durant l'exécution du programme. C'est lui qui assure l'intégrité de ces différents segments et qui empêche par exemple que la pile écrive dans le tas.
- S'il est besoin d'allouer plus de mémoire pour le tas que ce qui était prévu, c'est le mécanisme de *mémoire virtuelle* du système qui s'en charge.

Gestion de la mémoire par l'OS

- L'OS gère l'espace occupé par les segments durant l'exécution du programme. C'est lui qui assure l'intégrité de ces différents segments et qui empêche par exemple que la pile écrive dans le tas.
- S'il est besoin d'allouer plus de mémoire pour le tas que ce qui était prévu, c'est le mécanisme de *mémoire virtuelle* du système qui s'en charge.
- Il faut donner au programmeur l'impression que la RAM est infinie. Si la RAM vient à manquer, l'OS est capable d'utiliser automatiquement d'autres zones de stockage comme le disque dur ou une clé USB (évidemment, les performances se dégradent alors).

Remédiation en force brute

- On décide d'augmenter la taille de la pile :

```
$ ulimit -s unlimited
$ ulimit -s
unlimited
```

Cela permer d'augmenter la taille de la pile à la taille maximale du système d'exploitation. Sous LINUX, cette taille est illimité (mais l'administrateur peut imposer une limite). Sous MACOS, la limite est de 65 Mo.

Remédiation en force brute

- On décide d'augmenter la taille de la pile :

```
$ ulimit -s unlimited
$ ulimit -s
unlimited
```

  Cela permer d'augmenter la taille de la pile à la taille maximale du système d'exploitation. Sous LINUX, cette taille est illimité (mais l'administrateur peut imposer une limite). Sous MACOS, la limite est de 65 Mo.

- Il est maintenant possible d'exécuter le programme `./somme`. Mais l'exécution est lente du fait des empilements/dépilements successifs de stack frame.

Appel terminal

1. La *récursivité terminale* est une forme particulière de récursivité pouvant être optimisée afin de ne pas consommer de mémoire dans la pile.
2. Dans le corps d'une fonction, un appel est *terminal* s'il est la dernière opération effectuée par la fonction.
3. Les fonctions suivantes font un appel terminal à `g`

```
1 let f1 x = g x
2 let f2 x = if ... then g x else ...
3 let f3 x = let y = ... in g y
4 let f4 x = match x with
5       | ... -> ...
6       | ... -> g x
7       | _   -> ...
```

4. Appels à `g` non terminaux :

```
1 let f5 x = x + g x
2 let f6 x = let y = g x in y+1
```

Récursion terminale

1. Une fonction est dite *récursive terminale* si tous les appels récursifs dans sa définition sont en position terminale.

Récursion terminale

1. Une fonction est dite *récursive terminale* si tous les appels récursifs dans sa définition sont en position terminale.
2. Intérêt : il n'est plus nécessaire d'empiler les stack frame lors des appels récursifs. La stack frame de départ suffit.

Exemple

```
let rec f x = if x = 0 then 10 else f (x-1)
```

Gestion de la pile d'appel

- Pour `f 2` La case réservée pour la valeur retour est vide, l'argument `x` contient 2.

Pile
x : 2
ret. adr. : rien d'écrit

Pile
x : 1
ret. adr. : rien d'écrit

Pile
x : 0
ret. adr. : write 10

Exemple

```
let rec f x = if x = 0 then 10 else f (x-1)
```

Gestion de la pile d'appel

- Pour `f 2` La case réservée pour la valeur retour est vide, l'argument `x` contient 2.
- L'appel `f 1` peut utiliser la même stack frame que `f 2` car la valeur 2 contenue dans la case `x`, n'est plus utilisée par la suite. On met donc 1 dans `x`.

Pile
x : 2
ret. adr. : rien d'écrit

Pile
x : 1
ret. adr. : rien d'écrit

Pile
x : 0
ret. adr. : write 10

Exemple

```
let rec f x = if x = 0 then 10 else f (x-1)
```

Gestion de la pile d'appel

- Pour `f 2` La case réservée pour la valeur retour est vide, l'argument `x` contient 2.
- L'appel `f 1` peut utiliser la même stack frame que `f 2` car la valeur 2 contenue dans la case `x`, n'est plus utilisée par la suite. On met donc 1 dans `x`.
- L'appel `f 0` utilise encore la même stack frame en mettant 0 dans la case `x`. La valeur de retour (10) est mise dans `ret` car c'est la valeur qui est finalement renvoyée par le 1er appel (`f 2`).

Pile
x : 2
ret. adr. : rien d'écrit

Pile
x : 1
ret. adr. : rien d'écrit

Pile
x : 0
ret. adr. : write 10

Fonction somme : version en récursion terminale

- On utilise une fonction auxiliaire `sum` à deux paramètres : l'entier courant `x` et un *accumulateur* `acc`.

Fonction somme : version en récursion terminale

- On utilise une fonction auxiliaire `sum` à deux paramètres : l'entier courant `x` et un *accumulateur* `acc`.
- L'accumulateur grossit au fil des appels récursifs internes et il contient la valeur voulue lorsque `x` devient nul. On renvoie donc `acc`.

Fonction somme : version en récursion terminale

- On utilise une fonction auxiliaire `sum` à deux paramètres : l'entier courant `x` et un *accumulateur* `acc`.
- L'accumulateur grossit au fil des appels récursifs internes et il contient la valeur voulue lorsque `x` devient nul. On renvoie donc `acc`.
- Code :

```
1 let somme x =
2   let rec sum x acc = (*fonction auxiliaire*)
3     if x = 0 then acc else sum (x-1) (acc + x)
4   in sum x 0
5
6 let _ = let v = somme 1_000_000 in Printf.printf "%d" v
```

Fonction somme : version en récursion terminale

- On utilise une fonction auxiliaire `sum` à deux paramètres : l'entier courant `x` et un *accumulateur* `acc`.
- L'accumulateur grossit au fil des appels récursifs internes et il contient la valeur voulue lorsque `x` devient nul. On renvoie donc `acc`.
- Code :

```
1 let somme x =
2   let rec sum x acc = (*fonction auxiliaire*)
3     if x = 0 then acc else sum (x-1) (acc + x)
4   in sum x 0
5
6 let _ = let v = somme 1_000_000 in Printf.printf "%d" v
```

- D'une façon générale, une fonction auxiliaire est utile lorsqu'on a besoin de plus de paramètres que ceux initialement prévus. parfois aussi, on peut écrire écrire une fonction auxiliaire qui utilise moins de paramètres.

Fonction `somme` en C

- Il semble que les versions récentes de `ocamlopt` transforment automatiquement le code naaïf de `somme` en une version récursive terminale.

Fonction `somme` en C

- Il semble que les versions récentes de `ocamlopt` transforment automatiquement le code naaïf de `somme` en une version récursive terminale.
- Considérons

```
1  int somme (int n){
2    if (n==0)
3      return 0;
4    return n + somme (n-1);
5  }
6  int main(){
7    somme(1000000);
8    return 0;
9  }
10
```

Fonction `somme` en C

- Il semble que les versions récentes de `ocamlopt` transforment automatiquement le code naaïf de `somme` en une version récursive terminale.
- Considérons

```
1  int somme (int n){
2    if (n==0)
3      return 0;
4    return n + somme (n-1);
5  }
6  int main(){
7    somme(1000000);
8    return 0;
9  }
10
```

- `gcc somme.c` puis exécution : seg fault.

Fonction `somme` en C

- Considérons maintenant

```
1  int aux (int acc, int n){
2    if (n==0) return acc;
3    return aux (n+acc,n-1);
4  }
5  int somme2(int n){return aux(0,n);}
6  int main(){
7    somme2(1000000);
8    return 0;
9  }
10
```

Fonction `somme` en C

- Considérons maintenant

```
1  int aux (int acc, int n){
2    if (n==0) return acc;
3    return aux (n+acc,n-1);
4  }
5  int somme2(int n){return aux(0,n);}
6  int main(){
7    somme2(1000000);
8    return 0;
9  }
10
```

- `gcc somme.c` puis exécution : seg fault

Fonction `somme` en C

- Considérons maintenant

```
1  int aux (int acc, int n){
2    if (n==0) return acc;
3    return aux (n+acc,n-1);
4  }
5  int somme2(int n){return aux(0,n);}
6  int main(){
7    somme2(1000000);
8    return 0;
9  }
10
```

- `gcc somme.c` puis exécution : seg fault
- `gcc somme.c -O3` puis exécution : OK.