Paradigmes de programmation

Cours 3 : Pile et tas

Benoît Montagu — benoit.montagu@inria.fr

CC BY SA

Préparation à l'agrégation d'informatique — Automne 2022

Plan du cours

Au programme aujourd'hui :

- Pile et tas
- Appels et retours de fonction
- Passages de paramètres
- Quelques éléments d'allocation mémoire
- Appels terminaux

Pour introduire simplement ces concepts :

- Un petit langage de programmation, très simple : MINI
- Une machine abstraite avec pile et tas explicite

MINI : un petit langage de programmation

Le langage MINI : un petit langage impératif

Un petit language « *while* » avec références et appels de fonctions :

b	::=	$\mathbf{true} \mid \mathbf{false}$	(Booléens)
n	::=	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$	(Entiers)
ℓ	::=	$\ell_1 \mid \ell_2 \mid \ldots$	(Adresses)
v	::=	$() \mid b \mid n \mid \ell$	(Valeurs)
i	::=	$x \mid v$	(Immédiats)
a	::=	$i \mid i \boxplus i$	(Termes atomiques)
	$\mid$	$\mathbf{ref}\, i \mid !i \mid i := i$	
	$\mid$	$f(i, \ldots, i)$	
cmd	::=	a	(Commandes)
	$\mid$	$x = a$	
	$\mid$	$\mathbf{if}\, i\, \mathbf{then}\, \{\, cmds \,\}\, \mathbf{else}\, \{\, cmds \,\}$	
	$\mid$	$\mathbf{while}\, i\, \{\, cmds \,\}$	
$cmds$	::=	cmd^+	(Séquences de commandes)
$decl$	::=	$\mathbf{fun}\, f(x, \ldots, x) = \{\, cmds \,\}$	(Déclarations de fonction)
p	::=	$decl^+$	(Programmes)

Exemples de programme MINI (1)

```
fun fact_while(n) = {
  i = 1;
  r = 1;
  b = i < n;
  while b {
    i = i + 1;
    r = r * i;
    b = i < n
  };
  r
}

fun main() = {
  fact_while(5)
}
```

- Un programme *impératif* classique
- Les variables sont mutables

Exemples de programme MINI (2)

```
fun fact(n) = {
  b = n <= 0;
  if b
  then 1
  else {
    n1 = n - 1;
    p = fact(n1);
    n * p
  }
}

fun main () = {
  fact(5)
}
```

- Un programme *récursif* classique
- Ici aussi : les variables sont mutables mais elles ne sont écrites qu'une fois
- Car de nouvelles variables sont créées à chaque nouvel appel

Exemples de programme MINI (3)

```
fun swap(r1, r2) = {
  v1 = !r1;
  v2 = !r2;
  r1 := v2;
  r2 := v1
}

fun main() = {
  x1 = ref 1;
  x2 = ref 2;
  swap(x1, x2);
  !x1
}
```

- Un autre programme impératif classique
- Un programme qui *alloue* des références…
  … les modifie
  … et les déréférence

De quoi a-t-on besoin pour évaluer les programmes MINI ?

Pour MINI, on choisit *expressément* un modèle simple

- Quel est le code courant ?
  - Quel code est-on en train d'exécuter au sein de la fonction courante ?
  - Le code est une **liste de commandes** $cmds$
- Quelles sont les valeurs des variables courantes ?
  - On parle des variables relatives à l'exécution de la fonction courante
  - On utilise un **environnement** local : c-à-d une map finie E : variables → valeurs
- Que doit-on faire une fois que la fonction courante a terminé ?
  - Quelles fonctions étaient « en cours d'exécution » jusqu'ici ? (appels imbriqués)
  - Que doit-on faire du résultat calculé par la fonction courante ?
  - On utilise une **pile**
    C-à-d une liste π des actions à effectuer *par la suite*
    C'est ce qui est usuellement appelé la « pile d'appels »
- Vers quelles valeurs pointent les références ?
  - On utilise une **mémoire** globale
    C-à-d une map finie μ : adresses → valeurs
    C'est ce qui est usuellement appelé « tas »

Une machine abstraite pour évaluer les programmes MINI

- Un programme transforme une configuration en une autre :

$$\begin{array}{rcll} conf & ::= & (cmds, E, \pi, \mu) & \text{(Configuration)} \\ E & ::= & \cdot \mid E[x \mapsto v] & \text{(Environnement)} \\ \pi & ::= & \cdot \mid f : \pi & \text{(Pile d'appels)} \\ f & ::= & (x, cmds, E) \mid (\_, cmds, E) & \text{(Trame de pile)} \\ \mu & ::= & \cdot \mid \mu[\ell \mapsto v] & \text{(Mémoire)} \end{array}$$

- On va définir une relation d'évaluation *pas à pas* entre configurations :

  $conf_1 \longrightarrow conf_2$ se lit : « $conf_1$ se réduit en $conf_2$ »

  ℹ Vocabulaire : sémantique opérationnelle à petits pas
- Configuration initiale : $([main()], \cdot, \cdot, \cdot)$ (appel à la fonction *main*)
- Configuration finale : $([v], E, \cdot, \mu)$ (valeur obtenue, plus rien à faire ensuite)
- Un modèle simple :
  - Pas de pointeur vers la pile
  - Pas de registres
  - Pile et tas sont disjoints
  - Pile et tas sont structurés
  - Références allouées dans le tas
  - Variables locales allouées dans la pile

Évaluation « pas à pas » de la machine abstraite (1)

Évaluation des immédiats : $[\![i]\!]_E = \begin{cases} v & \text{si } i = v \\ E(x) & \text{si } i = x \end{cases}$

Conditionnelles :

$$\begin{array}{rcll} (\textbf{if } i \textbf{ then } \{ cmds_1 \} \textbf{ else } \{ cmds_2 \} :: cmds, E, \pi, \mu) & \longrightarrow & (cmds_1 +\!\!+ cmds, E, \pi, \mu) & \\ & & \text{si } [\![i]\!]_E = \textbf{true} & \\ (\textbf{if } i \textbf{ then } \{ cmds_1 \} \textbf{ else } \{ cmds_2 \} :: cmds, E, \pi, \mu) & \longrightarrow & (cmds_2 +\!\!+ cmds, E, \pi, \mu) & \\ & & \text{si } [\![i]\!]_E = \textbf{false} & \end{array}$$

Boucles :

$$\begin{array}{rcll} (\textbf{while } i \{ cmds' \} :: cmds, E, \pi, \mu) & \longrightarrow & (cmds' +\!\!+ \textbf{while } i \{ cmds' \} :: cmds, E, \pi, \mu) & \\ & & \text{si } [\![i]\!]_E = \textbf{true} & \\ (\textbf{while } i \{ cmds' \} :: cmds, E, \pi, \mu) & \longrightarrow & (() :: cmds, E, \pi, \mu) & \\ & & \text{si } [\![i]\!]_E = \textbf{false} & \end{array}$$

Évaluation « pas à pas » de la machine abstraite (2)

Production de valeurs :

$$\begin{array}{rcll} (i :: cmds, E, \pi, \mu) & \longrightarrow & (cmds, E, \pi, \mu) & \text{si } [\![i]\!]_E = () \text{ et } cmds \neq [] \\ (i_1 \boxplus i_2 :: cmds, E, \pi, \mu) & \longrightarrow & ([\![i_1]\!]_E \boxplus [\![i_2]\!]_E :: cmds, E, \pi, \mu) & \\ (\textbf{ref } i :: cmds, E, \pi, \mu) & \longrightarrow & (\ell :: cmds, E, \pi, \mu[\ell \mapsto [\![i]\!]_E]) & \text{avec } \ell \notin \operatorname{dom} \mu \\ (!i :: cmds, E, \pi, \mu) & \longrightarrow & (\mu(\ell) :: cmds, E, \pi, \mu) & \text{avec } [\![i]\!]_E = \ell \\ (i_1 := i_2 :: cmds, E, \pi, \mu) & \longrightarrow & (() :: cmds, E, \pi, \mu[\ell \mapsto [\![i_2]\!]_E]) & \text{avec } [\![i_1]\!]_E = \ell \end{array}$$

Évaluation « pas à pas » de la machine abstraite (3)

Assignations :

$$\begin{array}{rcll} (x = i :: cmds, E, \pi, \mu) & \longrightarrow & (() :: cmds, E[x \mapsto [\![i]\!]_E], \pi, \mu) & \\ (x = i_1 \boxplus i_2 :: cmds, E, \pi, \mu) & \longrightarrow & (() :: cmds, E[x \mapsto [\![i_1]\!]_E \boxplus [\![i_2]\!]_E], \pi, \mu) & \\ (x = \textbf{ref } i :: cmds, E, \pi, \mu) & \longrightarrow & (() :: cmds, E[x \mapsto \ell], \pi, \mu[\ell \mapsto [\![i]\!]_E]) & \text{avec } \ell \notin \operatorname{dom} \mu \\ (x = !i :: cmds, E, \pi, \mu) & \longrightarrow & (() :: cmds, E[x \mapsto \mu(\ell)], \pi, \mu) & \text{avec } [\![i]\!]_E = \ell \\ (x = i_1 := i_2 :: cmds, E, \pi, \mu) & \longrightarrow & (() :: cmds, E[x \mapsto ()], \pi, \mu[\ell \mapsto [\![i_2]\!]_E]) & \text{avec } [\![i_1]\!]_E = \ell \end{array}$$

Évaluation « pas à pas » de la machine abstraite (4)

Appels de fonctions :

Étant donné une déclaration : $\mathbf{fun}\, f(x_1, \ldots, x_n) = \{\ cmds'\ \}$

$$(x = f(i_1, \ldots, i_n) :: cmds, E, \pi, \mu) \longrightarrow (cmds', E', (x, cmds, E) : \pi, \mu)$$
$$(f(i_1, \ldots, i_n) :: cmds, E, \pi, \mu) \longrightarrow (cmds', E', (\_, cmds, E) : \pi, \mu)$$
$$\text{où } E' = [x_1 \mapsto [\![i_1]\!]_E, \ldots, x_n \mapsto [\![i_n]\!]_E]$$

ℹ Vocabulaire :

- Les x_k sont les **paramètres formels** de f (*formal parameters*)
- Les i_k sont les **paramètres réels** de f (*actual parameters*)
- **Appel par valeur** : les paramètres sont évalués avant l'appel
- **Passage de paramètres par copie** : les valeurs des paramètres sont copiées pour utilisation par l'appelé

Retours de fonctions :

$$([v], E, (x, cmds, E') : \pi, \mu) \longrightarrow ((\,) :: cmds, E'[x \mapsto v], \pi, \mu)$$
$$([v], E, (\_, cmds, E') : \pi, \mu) \longrightarrow (v :: cmds, E', \pi, \mu)$$

Exemples

- Implémentation de MINI en OCaml : lexer/parser + machine abstraite
- Code source disponible, bientôt sur `#prog`
- Programme `swap.mini`

```
$ dune exec -- ./cli.exe programs/swap.mini --interactive
```

- Programme `mc91.mini`

```
$ dune exec -- ./cli.exe programs/mc91.mini --interactive
```

- Exercice pour chez vous :
  - Quelle serait une spécification pour `mc91` ?
  - Preuve de correction ?
  - Preuve de terminaison ?

Passage de paramètres

Passages de paramètres (1)

En MINI, comme dans les 3 langages au programme (C, OCaml, Python) :

- Les paramètres des fonctions sont évalués *avant* l'appel de fonction
  ℹ C'est ce qu'on appelle « l'appel par valeur »
- Les paramètres des fonctions sont passés par copie
  ⚠ La copie est superficielle (on ne traverse pas les pointeurs)

En C :

- Si vous passez des `struct` en paramètres ils seront *copiés*
  Cela peut coûter cher si le `struct` est de grande taille
  Le style idiomatique en C est de passer un *pointeur* vers un `struct`
- Les pointeurs vers `struct` sont tellement courants en C
  qu'une syntaxe spéciale est introduite :
  pour `struct { int f } *p`, on peut écrire `p->f` à la place de `(*p).f`

❓ Que fait ce code ?

```c
void swap(int a, int b) {
  int tmp = b; b = a; a = tmp;
}
```

Passages de paramètres (2)

En Python :

- Ici encore : passage par copie
- Un objet est une « référence » : le « contenu » de l'objet n'est pas copié

```python
class Stack:
    def __init__(self):
        self.value = []

    def push(self, v):
        self.value.append(v)

    def get(self):
        return self.value

def push_twice(stack, v):
    stack.push(v)
    stack.push(v)


s = Stack()
s.push(42)
push_twice(s, 0)
print(s.get())
# quelle liste est affichée ?
```


Passages de paramètres (3)

En OCaml :

- Ici encore, passage par copie
- En pratique, le compilateur utilise une représentation uniforme des données : une valeur est représentée soit par un entier machine soit par un pointeur vers un bloc mémoire (vers le tas)

```
type stack = {
  value: int list ref
}

let init () = { value = ref [] }

let get s = !(s.value)

let push stack v =
  stack.value :=
    v :: !(stack.value)
```

```
let push_twice stack v =
  push stack v;
  push stack v

let l =
  let s = init () in
  push s 42;
  push_twice s 0;
  get s
  (* quel est le résultat ? *)
```


MINI : une simplification des machines réelles

Comparaison d'un état de MINI avec une machine réelle (1)

Code

- MINI : liste de commandes
- Machine réelle :
  - un pointeur vers l'instruction courante
  - code enregistré en mémoire, dans le segment `text` (*read-only*)
  - le pointeur de code est enregistré dans un *registre* (`pc`, ou `ip` suivant les architectures)

Comparaison d'un état de MINI avec une machine réelle (2)

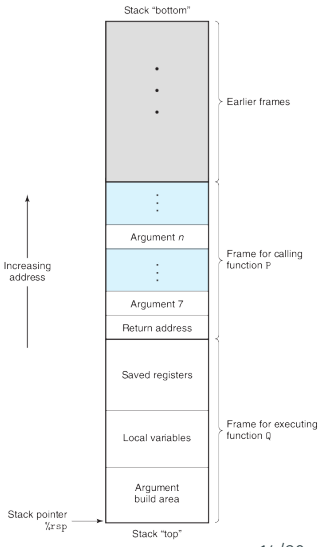
Variables locales :

- En MINI : environnement
- Machine réelle :
  - Données en registres
  - Données enregistrées dans la trame de pile courante (si besoin)
    (c'est la trame de pile située au sommet de la pile)

Comparaison d'un état de MINI avec une machine réelle (3)

Pile :

- MINI : liste de trames contenant variable de retour, environnement à restaurer, et code à restaurer
- Machine réelle :
  - La liste de trames est en mémoire : les trames les unes à côté des autres
  - Chaque trame contient :
    - les variables et données locales
    - les registres à restaurer
    - les arguments passés à la fonction courante
    - le pointeur de code à restaurer
    - et parfois : un « *canari* » (contre-mesure de sécurité)
  - Deux registres délimitent la trame courante (frame pointer et stack pointer)
  - Le contenu exact dépend de la *convention d'appel* de l'architecture, et du compilateur utilisé



Source : Computer Systems : A Programmer's Perspective (§3.7.1, Fig. 3.25)

Comparaison d'un état de MINI avec une machine réelle (4)

Tas :

- MINI : mémoire (map adresses → valeurs)
- Machine réelle :
  - Les adresses sont des entiers
  - Valeurs enregistrées en mémoire sous forme binaire
  - Tas et pile sont dans la même mémoire
    Adresses basses : tas Adresses hautes : pile
  - ⚠ Il faut se prémunir contre les chevauchements entre tas et pile
  - En général : l'OS préalloue l'espace pour la pile, et ajoute une « page de garde » juste avant

Point culturel : « *capability machines* »

- pointeur ≠ entier
- Impossibilité de « forger » des pointeurs
- Encore un sujet de recherche actif
- Architecture bientôt disponible ? (exemple : ARM-Morello/CHERI)

Pile : quelques informations supplémentaires

- Les constituants de la pile sont appelés :
  « stack frame » ou « activation record » ou « trame de pile »
- La trame la plus récente est appelée trame « courante »
  - Elle est au sommet de la pile
  - Elle est utilisée par la fonction qui est *en train* de s'exécuter
- Dans beaucoup de langages de programmation :
  - La fonction courante n'a accès qu'à la trame courante
  - Elle ne peut ni lire ni écrire dans les trames parentes
- En C : c'est possible, via un pointeur vers la pile parente

```c
void f(int *p) {
  *p = 0;
}

int main() {
  int x = 42; // x est alloué dans la pile de `main'
  f(&x);      // puis modifié par l'exécution de `f'
  return x;
}
```

Allocation de mémoire

Allocation : pile ou tas ?

Que contiennent pile et tas, pour les langages de programmation au programme ?

	Pile [1]	Tas
MINI	variable locales, paramètres	références
Python	variables locales, paramètres [2]	strings, objets, listes...
OCaml	variables locales, paramètres [2]	strings, clôtures, listes, objets, enregistrements, références...
C	variables locales, paramètres [3]	tout ce qui est créé par des appels à `malloc`/`calloc`

Et **toujours**, dans la pile : infos sur quoi faire au retour de la fonction courante (`pc`/`sp` de l'appelant, adresse pour la valeur à retourner, registres à restaurer...)

1. Ou dans les registres, si la place le permet
2. Généralement : ce qui peut tenir dans un registre (entiers, booléens, pointeurs...)
3. Tout ce qui est déclaré comme variable locale (y compris `struct`, `union`, tableaux...)

Exemples

En C :

```c
struct pair {
  int fst;
  int snd;
};

int f() {
  struct pair p;
  // le contenu de `p' est
  // alloué en pile
  p.fst = 17;
  p.snd = 25;
  int sum = p.fst + p.snd;
  // l'entier `sum' est
  // alloué en pile
  return sum;
}
```

En OCaml :

```ocaml
type pair = {
  fst: int;
  snd: int;
}

let f () =
  let p =
  (* le contenu de `p' est
     alloué dans le tas *)
    { fst = 17;
      snd = 25 } in
  let sum = p.fst + p.snd in
  (* l'entier `sum' est
     alloué en pile *)
  sum
```

(On suppose ici qu'aucune optimisation n'est effectuée)

Allocation : pile ou tas ? Avantages et inconvénients

- En Python et OCaml, vous n'avez pas le choix
  La mémoire (allocation/libération) est gérée automatiquement
- En C, vous pouvez choisir
  - Allocation en pile : par déclaration de variables locales
    - Aussi appelée « allocation statique »
    - ⊕ Libération automatique (au retour des fonctions)
    - ⊕ Allocation/libération : peu coûteux en temps (il suffit de modifier `sp`)
    - ⊖ La taille de la pile est limitée (et **beaucoup** plus petite que le tas [1])
    - ⊖ Tous les motifs de programmation ne se prêtent pas à une discipline d'allocation/libération qui soit *bien parenthésée*
  - Allocation dans le tas : par appels à `malloc`/`calloc`
    - Aussi appelée « allocation dynamique »
    - ⊕ La taille du tas est généralement grande [2]
    - ⊕ Offre plus de liberté dans le style de programmation
    - ⊖ Libération explicite de la mémoire (`free`) : difficile, source d'erreurs
    - ⊖ Allocation/libération : plus coûteux en temps (utilisation d'un allocateur mémoire)

1. Taille maximale de `stack` sous Linux : `ulimit -s` Sur ma machine : `8192 kbytes`
2. Taille maximale de `data` sous Linux : `ulimit -d` Sur ma machine : « `unlimited` »

Appels terminaux et consommation de pile

Appels terminaux

Exemples en MINI :

```
fun g(z) = { z * 2 }

fun f(x) = { y = x + 1; g(y) (* appel terminal *) }

fun main() = { f(20) (* un autre appel terminal *) }
```

```
$ # Exécutons pas à pas ce programme, avec la commande suivante:
$ dune exec -- ./cli.exe programs/tailcalls.mini --interactive
...
Main function: main
Result: 42
Maximum stack size: 3
Number of evaluation steps: 9
```


Optimisation d'appels terminaux pour la machine MINI

On ajoute un cas particulier à la relation de réduction lorsque
l'appel de fonction est la « dernière chose à faire pour la trame courante »

Étant donné une déclaration : $\textbf{fun}\, f(x_1, \dots, x_n) = \{\ cmds'\ \}$

$$([f(i_1, \dots, i_n)], E, \pi, \mu) \longrightarrow (cmds', E', \pi, \mu)$$
$$\text{où } E' = [x_1 \mapsto [\![i_1]\!]_E, \dots, x_n \mapsto [\![i_n]\!]_E]$$

Remarque : dans la commande $x = f(i_1, \dots, i_n)$, l'appel à f n'est pas terminal

Appels terminaux, avec machine abstraite optimisée

Les mêmes exemples en MINI :

```
fun g(z) = { z * 2 }

fun f(x) = { y = x + 1; g(y) (* appel terminal *) }

fun main() = { f(20) (* un autre appel terminal *) }
```

```
$ # Exécutons ce programme avec l'optimisation d'appels terminaux:
$ dune exec -- ./cli.exe programs/tailcalls.mini --tailcalls
...
Main function: main
Result: 42
Maximum stack size: 0
Number of evaluation steps: 6
```

👍 Cette fois, aucun espace de pile n'est consommé
ℹ Cela fonctionne aussi avec des fonctions récursives !

Appels terminaux en Python

En Python, les appels terminaux ne sont pas optimisés : ils consomment de la pile

```
>>> def loop(x, acc):
...     if (x <= 0):
...         return acc
...     return loop(x-1, acc+1)
...
>>> loop(997, 0)
997
>>> loop(998, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in loop
  File "<stdin>", line 4, in loop
  File "<stdin>", line 4, in loop
  [Previous line repeated 995 more times]
  File "<stdin>", line 2, in loop
RecursionError: maximum recursion depth exceeded in comparison
```


Appels terminaux en OCaml

En OCaml, les appels terminaux sont optimisés presque tout le temps

X. Leroy (`discuss.ocaml.org`, Octobre 2021) :

> *If there are many arguments to the call and they don't all fit in the processor registers reserved for argument passing, the remaining arguments are put on the stack, and a regular, non-tail call is performed. This limitation had been with us since day 1 of OCaml.*

Dans le même message, décrivant une correction pour la version `4.14` d'OCaml :

> *Once the registers available for passing function arguments are exhausted, the next 64 arguments are passed in a memory area that is part of the domain state. This argument passing is compatible with tail calls, so we get guaranteed tail calls up to 70 arguments at least. [...] Enjoy your 70-arguments tail calls!*

Appels terminaux en OCaml

```
(* Factorielle: version récursive, non terminale *)
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n-1)

(* Factorielle: version récursive, non terminale *)
let fact_tailrec =
  let rec aux acc n = (* `acc' est un accumulateur *)
    if n <= 0
    then acc
    else aux (n * acc) (n - 1)
  in
  aux 1
```

Exercice : démontrer que `fact` et `fact_tailrec` calculent les mêmes résultats

- ❓ Quel lemme auxiliaire est nécessaire ?
- ❓ Quelles propriétés de la multiplication sont nécessaires ?

Appels terminaux en C

En C, les appels terminaux *peuvent* être optimisés : cela dépend du compilateur, et du niveau d'optimisation choisi

```
#include <stdio.h>

// When compiled with -O2, tail-call optimization is performed
int loop(int x, int acc) {
  if (x <= 0) { return acc; }
  else { return loop(x-1, acc+1); }
}

int main() {
  int result = loop(500000, 0);
  printf("result = %d\n", result);
  return 0;
}
```

Pour en être certain, mieux vaut regarder le code assembleur produit !

```
$ gcc -std=c99 -S -O2 tailcall.c -o tailcall.S
```

Éléments pour raisonner sur la consommation de pile : en Python/OCaml

Comment déterminer la consommation de pile asymptotique dans le pire cas ?

- Python : sémantique peu claire pour ce genre d'analyse…
  De plus : la pile est celle de l'interpréteur, pas celle de la machine
  (le code est interprété : pas compilé)
- OCaml :
  - En 1re approximation : la taille des trames est proportionnelle au nombre de variables locales d'une fonction
  - Elle dépend du code et du compilateur
  - Elle ne dépend pas des valeurs des arguments des fonctions
  - Il suffit donc de compter le nombre d'appels de fonction qui sont imbriqués et non-terminaux pour avoir une approximation en $O(\cdot)$
- Si on insiste pour Python :
  - Faire comme en OCaml, en comptant cette fois les appels terminaux
- Dans tous les cas : si le sujet ne le précise pas,
  rappelez explicitement ce que vous comptez

Éléments pour raisonner sur la consommation de pile : en C

- **Problème :** à cause de l'allocation dans la pile,
  la taille des trames de pile peut dépendre des valeurs des arguments !

```c
int f(int n) {
  int t[n];     // un tableau de taille `n' est alloué en pile
  ...           // `t' est un VLA: Variable-Length Array
}
```

- Le support des VLAs est optionnel en C11
- Si le code n'utilise pas ce motif, alors la taille des frames ne dépend pas des arguments, et il suffit de compter les appels de fonctions imbriqués
- Mon conseil : *considérez* par défaut que
  les appels terminaux ne sont *pas* optimisés en C
- Dans tous les cas : si le sujet ne le précise pas,
  rappelez explicitement ce que vous comptez

Éléments pour raisonner sur la consommation de tas

- Python :
  - Taille des strings créées
  - Tuples, listes, ensembles, objets
  - La taille des listes/tableaux/strings peut dépendre de la valeur des paramètres
  - Taille des entiers calculés (ils sont en précision arbitraire !)
- OCaml :
  - Taille des strings créées
  - Tuples, variants (constructeurs), records, objets, clôtures, `lazy`
  - La taille des listes/tableaux/strings peut dépendre de la valeur des paramètres
- C :
  - Appels à `malloc`/`calloc`
  - L'argument peut dépendre de la valeur des paramètres

Conclusion

Pour aller plus loin (1)

Randal E. Bryant et David R. O'Hallaron (fév. 2018). Computer Systems: A Programmer's Perspective, Global Edition. Pearson. 1120 p. ISBN : 1292101768

Chapitre 3 : Machine-Level Representation of Programs

- Section 7 (Procedures) : explications détaillées de gestion de la pile pour architecture `x86-64`
- Autres sections : beaucoup d'autres explications sur comment représenter des types et des structures de contrôle de C en machine

Pour aller plus loin (2)

Michael Scott (déc. 2015). Programming Language Pragmatics. Fourth Edition. San Francisco, CA : Morgan Kaufmann Pub. 992 p. ISBN : 9780124104099

Chapitre 8 : Subroutines and Control Abstraction

- Autres modes de passage de paramètres, pour d'autres langages
- Éléments de réponse pour implémenter les exceptions, les coroutines

Pour aller plus loin (3)

Sylvain Conchon et Jean-Christophe Filliâtre (sept. 2014). Apprendre à programmer avec Ocaml: Algorithmes et structures de données. ADIZES INST. 444 p. ISBN : 2212136781

Chapitre 3 : Approfondir les concepts d'OCaml

- Section 2 (Modèle d'exécution) : en particulier, représentation des données (représentation uniforme)

Conclusion

- Un modèle d'exécution simple pour un petit langage (Mini)
  - Une pile pour gérer les appels de fonction
  - Un tas pour enregistrer les valeurs des références
- Lien entre ce modèle et une machine réelle
- Utilisation de la pile et du tas pour les langages au programme
- Quelques remarques pour raisonner sur la consommation de mémoire
- Retrouvez le code de l'interpréteur Mini sur `Discord`

Cet après-midi :

Programmation fonctionnelle