Un Assembleur en OCamL
Matthieu Boyer
Planche A

Khôlle MP2I
11/11/2024

Pointeurs sur un modèle de RAM

On donne la représentation suivante d'une mémoire vive :

```
type bit = Zero | One;;
type address = bit array;;
type ram = bit array;;
```

Ici, on représente la RAM comme un tableau binaire. On stockera le pointeur vers le début de la stack sur le premier octet de la RAM. On supposera que tous nos entiers se représentent sur 8 bits en petit-boutiste.

1. Donnez une fonction `init: ram` qui crée une RAM vide.
2. Donnez une fonction `référencer: int -> ram -> address` qui a un entier et une ram stocke l'entier dans la ram et renvoie un pointeur vers l'entier. On agira sur la ram avec effet de bord.
3. Donnez une fonction `modifier: address -> int -> ram -> unit` qui a un pointeur et un entier modifie la valeur de l'entier associée au pointeur.
4. Donnez une fonction `déréférencer: ram -> address -> int` qui a une ram et un pointeur et renvoie la valeur de l'entier associé au pointeur.
5. Définissez une fonction `arith: address -> int -> address` telle que `arith p n` renvoie le pointeur vers l'addresse définie par $p + n$.
6. Donnez une fonction `free: address -> ram -> unit` qui libère la variable associée au pointeur.

Solution: Pointeurs sur un modèle de RAM

On commence par redonner une fonction de conversion décimal vers binaire et binaire vers décimal.

```
let nth_bit x n = x land (1 lsl n) <> 0;;
let to_bin x = Array.init 8 (nth_bit x);;
let to_dec barray = Array.fold_left (fun acc x -> acc + x) 0
                (Array.mapi (fun i -> barray.(i) * (pow 2 i)) barray);;
```

1.
```
let init =
        let r = Array.make (256 * 8) Zero in r.(0) <- 1;
r;;
```

2.
```
let référencer n r =
        let stack_top = (to_dec (Array.sub r 0 8)) in
        let bit_n = to_bin n in
        for i = 0 to 8 do
                r.(8 * stack_top + i) <- bit_n.(i)
        done;
        let p = to_bin (stack_top + 1) in
        for i = 0 to 8 do r.(i) <- p.(i) done;
        p;;
```

3.
```
let modifier p n r =
        let ad = to_dec p in
        let bit_n = to_bin n in
        for i = 0 to 8 do
                r.(8 * to_dec + i) <- bit_n.(i)
        done;;
```

4.
```
let déréférencer p r = let ad = to_dec p in to_dec (Array.sub r ad (ad + 8));;
```

5.
```
let arith p i = to_bin ((to_dec p) + i);;
```

6.
```
let free p r =
        let stack_top = (to_dec (Array.sub r 0 8)) in
        let free_address = to_dec p in
        let len = 8 * (stack_top - free_address) in
        let values = Array.sub r (8 * (free_address + 1)) (8*(stack_top + 1)) in
        Array.blit values 0 r (8 * free_address) len;
(*Array.blit équivaut à une boucle simple de copie élément par élément*)
        let p = to_bin (stack_top - 1) in
        for i = 0 to 8 do r.(i) <- p.(i) done;;
```

Un Assembleur en OCamL
Matthieu Boyer
Planche B

Khôlle MP2I
11/11/2024

Deux Bouts

On définit le type suivant :

```
type 'a elem = {val: 'a; mutable prev = 'a elem option; mutable next = 'a elem option};;
type 'a dll = {mutable first: 'a node option; mutable last: 'a node option};;
```

On rappelle que le type option est défini par : `type 'a option = None | Some of 'a` . Le type `dll` permet de représenter une forme de listes appelées doublement chaînées à deux bouts ; c'est à dire pour lesquelles, on peut de plus accéder au premier et au dernier élément.

1. Donnez une fonction `node: 'a -> 'a elem` .
2. Donnez une fonction `create: unit -> 'a dll` d'initialisation d'une liste vide.
3. Donnez une fonction `append: 'a dll -> 'a -> unit` d'ajout d'un élément à la fin d'une liste.
4. Donnez une fonction `drop: 'a dll -> 'a elem -> unit` qui supprime un élément d'une liste.

On s'intéresse désormais aux files à deux bouts, c'est à dire une structure de données où on peut accéder aux premier et dernier élément, et où on peut ajouter un élément au début comme à la fin. On donne le type suivant pour représenter cette structure : `type 'a de_list = 'a list * 'a list` . Ici, la liste complète est la concaténation de la première list F et du miroir de la deuxième R.

5. Écrivez les fonctions `cons: 'a -> 'a de_list -> 'a de_list` qui ajoute au début et `snoc: 'a de_list -> 'a -> 'a de_list` qui ajoute à la fin.
6. Si c est un entier, proposez une méthode pour qu'on ait $|F| < c\,|R| + 1$ et $|R| < |F| + 1$. On pourra utiliser une fonction `take` qui renvoie les i premiers éléments d'une liste.

Solution: Deux Bouts

1. `let node x = {value = x; prev = None; next = None}`
2. `let create () = {first = None; last = None}`
3.
```
let append t n =
        match t.last with
        |None -> let node = Some n in
                 t.first <- node; t.last <- node
        | Some lst -> let node = Some n in
                 lst.next <- node;
                 n.prev <- t.last;
                 t.last <- node
```
4.
```
let drop t n =
        let np = n.prev in
        let nn = n.next in
        (match np with
                 | None -> t.first <- nn;
                 | Some x -> x.next <- nn; n.prev <- None
        );
        match nn with
                 | None -> t.last <- np
                 | Some x -> x.prev <- np; n.next -> None
```
5.
```
let cons n nl = (n::(fst nl), snd nl);;
let snoc nl n = (fst nl, n::(snd nl));;
```

Si on souhaitait être plus efficaces, on pourrait équilibrer au fur et à mesure la longueur des deux parties, par exemple comme décrit dans *Purely Functional Data Structures* de Chris Okasaki Section 5.4 (page 56).

Un Assembleur en OCamL
Matthieu Boyer
Planche C

Khôlle MP2I
11/11/2024

La Paresse

On se propose d'implémenter en OCamL une bibliothèque de calcul paresseux. Celle-ci expose un type paramétré de suspensions `'a susp` et des fonctions de types suivants :

— `susp: (unit -> 'a) -> 'a susp`

— `force: 'a susp -> 'a`

Une suspension (de type `'a susp`) contient une fonction permettant de calculer une valeur de type `'a`. Elle peut être construite facilement grâce à la fonction `susp`. Le calcul n'est effectué que lorsque l'utilisateur de la bibliothèque le demande via la fonction `force`. Cette dernière fonction vérifie si le calcul a déjà été effectué : si tel est le cas, elle en renvoie le résultat pré-calculé. Sinon, elle lance le calcul, stocke le résultat pour de futurs appels, et elle le renvoie.

1. Donner une implémentation possible de cette bibliothèque, dans le langage OCamL.

On définit le type des *listes paresseuses* en OCamL :

```
type 'a slist_cell =
| SNil
| SCons of 'a * 'a slist
and 'a slist = 'a slist_cell susp;;
```

2. (a) Comparer la notion de liste paresseuse avec la notion habituelle de liste.
   (b) Écrire une fonction `scons: 'a -> 'a slist -> 'a slist` qui ajoute un élément en tête d'une liste paresseurse, ainsi qu'une valeur `snil` de liste paresseuse vide.
   (c) Écrire deux fonctions `shd: 'a slist -> 'a` et `stl: 'a slist -> 'a slist` qui prennent une liste paresseuse non vide en paramètre et qui renvoie respectivement son premier élément et la liste paresseuse des autres éléments.
   (d) Écrire une fonction `sappend: 'a slist -> 'a slist -> 'a slist` qui concatène de manière paresseuse deux listes paresseuses. Cette fonction devra s'exécuter en temps constant.
   (e) Écrire une fonction `srev! 'a slist -> 'a slist` qui renverse une liste paresseuse de manière efficace. Quelle est la complexité des accès aux différents éléments de la liste renversée ?

Solution: La Paresse

1. Une suspension peut avoir deux états : en attente de calcul ou calculé. Cet état est modifié de manière impérative lors du premier appel à `force` la concernant. Il est donc naturel d'utiliser un type somme à deux alternatives pour représenter une suspension. De plus, pour permettre le changement d'état, on va utiliser une référence :

```
type 'a susp_state =
| Computed of 'a
| Suspended of (unit -> 'a)

type 'a susp = 'a susp_state ref
```

Pour la fonction `susp` on utilise simplement le constructeur `Suspended` dans une référence fraîche : `let susp f = ref (Suspended f)` Enfin, la fonction `force` peut être implémentée facilement en distinguant les deux cas :

```
let force p =
        match !p with
        | Computed x -> x
        | Suspended f ->
                let x = f () in
                p := Computed x;
                x
```

2. Dans la notion de liste habituelle, les éléments de la liste sont présents en mémoire dès que la liste existe. Au contraire, pour une liste paresseuse, ils sont calculés à la demande, lorsque l'on accède aux éléments de la liste. De manière intéressante, la structure de liste paresseuse peut être utilisée pour représenter des séquences infinies d'éléments, en ne stockant pas explicitement les éléments qui ne sont pas encore accédés : on stocke plutôt une fonction qui permettra, le moment venu, de calculer plus d'éléments de cette liste.

```
let snil = susp (fun () -> SNil)
let scons x l = susp (fun () -> SCons (x, l))

let shd x =
        match force x with
        | SCons (hd, _) -> hd
        | SNil -> assert false
let stl x =
        match force x with
        | SCons (_, tl) -> tl
        | SNil -> assert false

let rec sappend l1 l2 =
        susp (fun () ->
                match force l1 with
                | SCons (t, q) -> SCons (t, sappend q l2)
                | SNil -> force l2)
```