Bitsets_en_OCaml

May 21, 2019

1 Table of Contents

2 BitSets en OCaml

Let *BitSets* sont une implémentation efficace pour représenter, stocker et manipuler des ensembles de *petits* entiers.

Avec un seul entier 32 bits, on peut représenter *tous* les ensembles d'entiers de $\{0, \ldots, 31\}$.

On utilise les opérations bit à bit ("bitwise") de OCaml pour effectuer les opérations de base sur les ensembles.

Cette implémentation est inspiré de celle ci.

2.1 Type abstrait

Je vais faire ça à l'arrache, mais on pourrait faire soit un module qui contient un type interne (caché), soit avec un champ d'enregistrement `{ n : int }`.

Cette approche est la plus simple, mais montre un peu le fonctionnement interne (on pourrait vouloir l'éviter).

```
In [1]: type bit_set_32 = int;;
```

```
In [2]: let max_size_bit_set_32 = 32;;
```

Avec Int64 on pourrait faire la même chose avec des entiers sur 64 bits. La flemme.

2.2 Implémentation

Les ensembles ne seront pas mutables dynamiquement : comme une liste, toute fonction qui modifie un `bit_set_32` renvoit un nouveau `bit_set_32`.

```
In [3]: let empty () : bit_set_32 = 0
        ;;
```

Le singleton $\{i\}$ est codé comme 2^i donc 1 au bit en $i$ème position. En Python, c'est `1 << i` (ou `2**i`), et en OCaml c'est `1 lsl i`.

```
In [4]: let create (i : int) : bit_set_32 =
            assert ((0 <= i) && (i < max_size_bit_set_32));
            1 lsl i
        ;;

Out[4]: val create : int -> bit_set_32 = <fun>
```

Si on voulait utiliser la syntaxe Python, on pourrait faire ça :

```
In [5]: let ( << ) = ( lsl );;
        let ( >> ) = ( lsr );;

Out[5]: val ( << ) : int -> int -> int = <fun>


Out[5]: val ( >> ) : int -> int -> int = <fun>
```

La copie ne fait pas beaucoup de sens comme la stucture n'est pas mutable… Mais soit.

```
In [6]: let copy (b : bit_set_32) : bit_set_32 =
            b + 0 (* enough ? *)
        ;;
        let clone = copy;;

Out[6]: val copy : bit_set_32 -> bit_set_32 = <fun>


Out[6]: val clone : bit_set_32 -> bit_set_32 = <fun>
```

`set b n` permet d'ajouter `n` dans l'ensemble `b`, et `unset b n` permet de l'enlever (peu importe s'il était présent ou non).

```
In [7]: let set (b : bit_set_32) (n : int) : bit_set_32 =
            b lor (1 lsl n)
        ;;

        let add = set;;
```

```
Out[7]: val set : bit_set_32 -> int -> bit_set_32 = <fun>


Out[7]: val add : bit_set_32 -> int -> bit_set_32 = <fun>


In [8]: let unset (b : bit_set_32) (n : int) : bit_set_32 =
            b land (lnot (1 lsl n))
        ;;

        let remove = unset;;

Out[8]: val unset : bit_set_32 -> int -> bit_set_32 = <fun>


Out[8]: val remove : bit_set_32 -> int -> bit_set_32 = <fun>
```

On peut combiner `set` et `unset` pour faire :

```
In [9]: let put (b : bit_set_32) (p : bool) (n : int) : bit_set_32 =
            if p then set b n else unset b n
        ;;

Out[9]: val put : bit_set_32 -> bool -> int -> bit_set_32 = <fun>
```

Ces deux autres opérations sont rapides :

```
In [10]: let is_set (b : bit_set_32) (n : int) : bool =
             let i = (b land (1 lsl n)) lsr n in
             i = 1
         ;;

         let contains_int = is_set;;
         let is_in = is_set;;

Out[10]: val is_set : bit_set_32 -> int -> bool = <fun>


Out[10]: val contains_int : bit_set_32 -> int -> bool = <fun>


Out[10]: val is_in : bit_set_32 -> int -> bool = <fun>


In [11]: let toggle (b : bit_set_32) (n : int) : bit_set_32 =
             put b (not (is_set b n)) n
         ;;
```

```
Out[11]: val toggle : bit_set_32 -> int -> bit_set_32 = <fun>
```

La comparaison et le test d'égalité sont évidents.

```
In [12]: let compare (b1 : bit_set_32) (b2 : bit_set_32) : int =
             Pervasives.compare b1 b2
         ;;

Out[12]: val compare : bit_set_32 -> bit_set_32 -> int = <fun>
```

```
In [13]: let equals (b1 : bit_set_32) (b2 : bit_set_32) : bool =
             b1 = b2
         ;;

Out[13]: val equals : bit_set_32 -> bit_set_32 -> bool = <fun>
```

On peut chercher à être plus efficace que cette implémentation impérative et naïve. Essayez !

```
In [14]: let count (b : bit_set_32) : int =
             let s = ref 0 in
             for n = 0 to max_size_bit_set_32 - 1 do
                 if is_set b n then incr s
             done;
             !s
         ;;

         let length = count;;

Out[14]: val count : bit_set_32 -> int = <fun>
```

```
Out[14]: val length : bit_set_32 -> int = <fun>
```

Pour la création des exemples, on va aussi se permettre de convertir un `bit_set_32` en une `int list`, et inversement de créer un `bit_set_32` depuis une `int list`.

```
In [15]: let bit_set_32_to_list (b : bit_set_32) : (int list) =
             let list_of_b = ref [] in
             for n = 0 to max_size_bit_set_32 - 1 do
                 if is_set b n then
                 list_of_b := n :: !list_of_b
             done;
             List.rev (!list_of_b)
         ;;

Out[15]: val bit_set_32_to_list : bit_set_32 -> int list = <fun>
```

```
In [16]: let bit_set_32_from_list (list_of_b : int list) : bit_set_32 =
             let b = ref (empty()) in
             List.iter (fun i -> b := add !b i) list_of_b;
             !b
         ;;

Out[16]: val bit_set_32_from_list : int list -> bit_set_32 = <fun>


In [17]: let bit_set_32_iter (f : int -> unit) (b : bit_set_32) : unit =
             List.iter f (bit_set_32_iter_to_list b)
         ;;


         File "[17]", line 2, characters 17-40:
    Error: Unbound value bit_set_32_iter_to_list
       1: let bit_set_32_iter (f : int -> unit) (b : bit_set_32) : unit =
       2:     List.iter f (bit_set_32_iter_to_list b)
       3: ;;
```

Pour l'affichage des exemples, on va aussi se permettre de convertir un `bit_set_32` en une `string`.

```
In [18]: let bit_set_32_to_string (b : bit_set_32) : string =
             let list_of_b = bit_set_32_to_list b in
             match List.length list_of_b with
             | 0 -> "{}"
             | 1 -> "{" ^ (string_of_int (List.hd list_of_b)) ^ "}"
             | _ -> begin
                 let s = ref ("{" ^ (string_of_int (List.hd list_of_b))) in
                 List.iter (fun i -> s := !s ^ ", " ^ (string_of_int i)) (List.tl list_of_b);
                 !s ^ "}"
             end
         ;;

Out[18]: val bit_set_32_to_string : bit_set_32 -> string = <fun>


In [19]: let print_bit_set_32 (b : bit_set_32) : unit =
             print_endline (bit_set_32_to_string b)
         ;;

Out[19]: val print_bit_set_32 : bit_set_32 -> unit = <fun>
```

Comme le domaine est fixé (à $\{0,\ldots,31\}$), on peut prendre le complémentaire.

```
In [20]: let complementaire (b : bit_set_32) : bit_set_32 =
             lnot b
         ;;

Out[20]: val complementaire : bit_set_32 -> bit_set_32 = <fun>
```

Les opérations intersection, union, différence et différence symétrique sont très faciles.

```
In [21]: let intersection (b1 : bit_set_32) (b2 : bit_set_32) : bit_set_32 =
             b1 land b2
         ;;

Out[21]: val intersection : bit_set_32 -> bit_set_32 -> bit_set_32 = <fun>


In [22]: let union (b1 : bit_set_32) (b2 : bit_set_32) : bit_set_32 =
             b1 lor b2
         ;;

Out[22]: val union : bit_set_32 -> bit_set_32 -> bit_set_32 = <fun>
```

Avec l'union on peut facilement tester si `b1` est contenu dans `b2` ($b_1 \subset b_2 \equiv (b_1 \cup b_2) = b_2$)

```
In [23]: let contains (b1 : bit_set_32) (b2 : bit_set_32) : bool =
             (union b1 b2) = b2
         ;;

Out[23]: val contains : bit_set_32 -> bit_set_32 -> bool = <fun>


In [24]: let difference (b1 : bit_set_32) (b2 : bit_set_32) : bit_set_32 =
             intersection b1 (complementaire b2)
         ;;

Out[24]: val difference : bit_set_32 -> bit_set_32 -> bit_set_32 = <fun>


In [25]: let difference_sym (b1 : bit_set_32) (b2 : bit_set_32) : bit_set_32 =
             b1 lxor b2
         ;;

Out[25]: val difference_sym : bit_set_32 -> bit_set_32 -> bit_set_32 = <fun>
```

2.3 Exemples

```
In [26]: print_bit_set_32 (empty());;
```

```
{}
```

```
Out[26]: - : unit = ()
```

```
In [27]: let b1 = bit_set_32_from_list [0; 12]
         and b2 = bit_set_32_from_list [1; 3; 6]
         and b3 = bit_set_32_from_list [0; 3; 6; 17]
         ;;

         print_bit_set_32 b1;;
         print_bit_set_32 b2;;
         print_bit_set_32 b3;;
```

```
Out[27]: val b1 : bit_set_32 = 4097
         val b2 : bit_set_32 = 74
         val b3 : bit_set_32 = 131145
```

```
{0, 12}
```

```
Out[27]: - : unit = ()
```

```
{1, 3, 6}
```

```
Out[27]: - : unit = ()
```

```
{0, 3, 6, 17}
```

```
Out[27]: - : unit = ()
```

2.3.1 Tests des opérations unaires

Tester l'appartenance :

```
In [28]: (is_in b1 3);;
         (is_in b2 3);;
         (is_in b3 3);;
```

```
Out[28]: - : bool = false


Out[28]: - : bool = true


Out[28]: - : bool = true


In [29]: (is_in b1 0);;
         (is_in b2 0);;
         (is_in b3 0);;

Out[29]: - : bool = true


Out[29]: - : bool = false


Out[29]: - : bool = true
```

On peut ajouter une valeur :

```
In [30]: print_bit_set_32 (add b1 20);;
         print_bit_set_32 (add b2 20);;
         print_bit_set_32 (add b3 20);;

{0, 12, 20}


Out[30]: - : unit = ()


{1, 3, 6, 20}


Out[30]: - : unit = ()


{0, 3, 6, 17, 20}


Out[30]: - : unit = ()
```

Ou l'enlever :

```
In [31]: print_bit_set_32 (remove b1 3);;
         print_bit_set_32 (remove b2 3);;
         print_bit_set_32 (remove b3 3);;
```

```
{0, 12}

Out[31]: - : unit = ()

{1, 6}

Out[31]: - : unit = ()

{0, 6, 17}

Out[31]: - : unit = ()

In [32]: length b1;;
         length b2;;
         length b3;;

Out[32]: - : int = 2

Out[32]: - : int = 3

Out[32]: - : int = 4

In [33]: print_bit_set_32 (complementaire b1);;
         print_bit_set_32 (complementaire b2);;
         print_bit_set_32 (complementaire b3);;

         print_bit_set_32 (complementaire (union (union b1 b2) b3));;

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27

Out[33]: - : unit = ()

{0, 2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 1

Out[33]: - : unit = ()

{1, 2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 1
```

```
Out[33]: - : unit = ()


{2, 4, 5, 7, 8, 9, 10, 11, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,


Out[33]: - : unit = ()
```

2.3.2 Tests des opérations binaires

```
In [34]: print_bit_set_32 (union b1 b2);;
         print_bit_set_32 (union b1 b3);;
         print_bit_set_32 (union b2 b3);;

Out[34]: - : unit = ()


Out[34]: - : unit = ()


{0, 1, 3, 6, 12}
{0, 3, 6, 12, 17}
{0, 1, 3, 6, 17}


Out[34]: - : unit = ()


In [35]: contains b1 b2;;
         contains b1 b3;;
         contains b1 (intersection b1 b3);;
         contains (intersection b1 b3) b1;;
         contains b1 (union b1 b3);;
         contains b2 b3;;

Out[35]: - : bool = false


Out[35]: - : bool = false


Out[35]: - : bool = false


Out[35]: - : bool = true


Out[35]: - : bool = true
```

```
Out[35]: - : bool = false


In [36]: print_bit_set_32 (intersection b1 b2);;
         print_bit_set_32 (intersection b1 b3);;
         print_bit_set_32 (intersection b2 b3);;

{}


Out[36]: - : unit = ()


{0}


Out[36]: - : unit = ()


{3, 6}


Out[36]: - : unit = ()


In [37]: print_bit_set_32 (difference b1 b2);;
         print_bit_set_32 (difference b1 b3);;
         print_bit_set_32 (difference b2 b3);;

{0, 12}


Out[37]: - : unit = ()


{12}


Out[37]: - : unit = ()


{1}


Out[37]: - : unit = ()


In [38]: print_bit_set_32 (difference_sym b1 b2);;
         print_bit_set_32 (difference_sym b1 b3);;
         print_bit_set_32 (difference_sym b2 b3);;
```

```
{0, 1, 3, 6, 12}

Out[38]: - : unit = ()

{3, 6, 12, 17}

Out[38]: - : unit = ()

{0, 1, 17}

Out[38]: - : unit = ()
```

2.4 Comparaison

On va essayer de comparer notre implémentation avec une implémentation naïve utilisant des `bool array` et une utilisant le module `Set` de la bibliothèque standard.

2.4.1 Mesure un temps d'éxecution

```
In [132]: let time (n : int) (f : unit -> 'a) : float =
              let t = Sys.time() in
              for _ = 0 to n-1 do
                  ignore (f ());
              done;
              let delta_t = Sys.time() -. t in
              let t_moyen = delta_t /. (float_of_int n) in
              Printf.printf "    Temps en secondes: %fs\n" t_moyen;
              flush_all ();
              t_moyen
          ;;

Out[132]: val time : int -> (unit -> 'a) -> float = <fun>

In [43]: Random.self_init();;

         let random_int_0_31 () =
             Random.int 32
         ;;

Out[43]: - : unit = ()

Out[43]: val random_int_0_31 : unit -> int = <fun>
```

2.4.2 Suite de test pour `bit_set_32`

Notre test va consister à créer un ensemble vide, et ajouter 100 fois de suite des valeurs aléatoires, en enlever d'autres etc.

```
In [50]: let test_bit_set_32 n n1 n2 () =
             let b = ref (empty ()) in
             for _ = 0 to n do
                 let nb_ajout = Random.int n1 in
                 let nb_retrait = Random.int n2 in
                 for i = 0 to nb_ajout + nb_retrait do
                     let n = random_int_0_31 () in
                     if i mod 2 = 0 then
                         b := add !b n
                     else
                         b := remove !b n;
                 done
             done;
             length !b
         ;;

Out[50]: val test_bit_set_32 : int -> int -> int -> unit -> int = <fun>


In [91]: test_bit_set_32 100 20 10 ();;

Out[91]: - : int = 13
```

2.4.3 Suite de test pour `bool array`

Avec des `bool array`, on a l'avantage d'avoir une structure dynamique.

```
In [99]: let test_boolarray n n1 n2 () =
             let b = Array.make max_size_bit_set_32 false in
             for _ = 0 to n do
                 let nb_ajout = Random.int n1 in
                 let nb_retrait = Random.int n2 in
                 for i = 0 to nb_ajout + nb_retrait do
                     let n = random_int_0_31 () in
                     if i mod 2 = 0 then
                         b.(n) <- true
                     else
                         b.(n) <- false
                 done;
             done;
             Array.fold_left (+) 0 (Array.map (fun x -> if x then 1 else 0) b)
         ;;
```

```
Out[99]: val test_boolarray : int -> int -> int -> unit -> int = <fun>


In [127]: test_boolarray 100 20 10 ();;

Out[127]: - : int = 15
```

2.4.4 Suite de test pour `Set.Make(Int)`

```
In [116]: module Int = struct
            type t = int
            let compare = compare
          end;;

          module Int32Set = Set.Make(Int);;

Out[116]: module Int :
            sig type t = int val compare : bit_set_32 -> bit_set_32 -> int end


Out[116]: module Int32Set :
            sig
              type elt = Int.t
              type t = Set.Make(Int).t
              val empty : t
              val is_empty : t -> bool
              val mem : elt -> t -> bool
              val add : elt -> t -> t
              val singleton : elt -> t
              val remove : elt -> t -> t
              val union : t -> t -> t
              val inter : t -> t -> t
              val diff : t -> t -> t
              val compare : t -> t -> int
              val equal : t -> t -> bool
              val subset : t -> t -> bool
              val iter : (elt -> unit) -> t -> unit
              val map : (elt -> elt) -> t -> t
              val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
              val for_all : (elt -> bool) -> t -> bool
              val exists : (elt -> bool) -> t -> bool
              val filter : (elt -> bool) -> t -> t
              val partition : (elt -> bool) -> t -> t * t
              val cardinal : t -> int
              val elements : t -> elt list
              val min_elt : t -> elt
              val max_elt : t -> elt
              val choose : t -> elt
```

```
  val split : elt -> t -> t * bool * t
  val find : elt -> t -> elt
  val of_list : elt list -> t
end
```

Avec des `Set`, on a l'avantage d'avoir une structure dynamique, mais moins facile à manipuler.

```
In [125]: let test_Int32Set n n1 n2 () =
              let b = ref (Int32Set.empty) in
              for _ = 0 to n do
                  let nb_ajout = Random.int n1 in
                  let nb_retrait = Random.int n2 in
                  for i = 0 to nb_ajout + nb_retrait do
                      let n = random_int_0_31 () in
                      if i mod 2 = 0 then
                          b := Int32Set.add n !b
                      else
                          b := Int32Set.remove n !b
                  done;
              done;
              Int32Set.cardinal !b
          ;;

Out[125]: val test_Int32Set : int -> int -> int -> unit -> int = <fun>


In [126]: test_Int32Set 100 20 10 ();;

Out[126]: - : int = 14
```

2.4.5 Mesurer les temps d'exécution

On va faire 500 répétitions de ces tests aléatoires, chacun avec 1000 fois des ajouts de 30 valeurs et des retraits de 20 valeurs.

```
In [134]: time 500 (test_bit_set_32 1000 30 20);;

    Temps en secondes: 0.004636s


Out[134]: - : float = 0.00463642199999999768


In [135]: time 500 (test_boolarray 1000 30 20);;

    Temps en secondes: 0.004257s
```

```
Out[135]: - : float = 0.00425716600000000146
```

```
In [136]: time 500 (test_Int32Set 1000 30 20);;

    Temps en secondes: 0.011854s
```

```
Out[136]: - : float = 0.0118542480000000013
```

Pour un second et dernier test, on va faire 500 répétitions de ces tests aléatoires, chacun avec 500 fois des ajouts de 100 valeurs et des retraits de 110 valeurs.

```
In [140]: time 500 (test_bit_set_32 500 100 110);;

    Temps en secondes: 0.009721s
```

```
Out[140]: - : float = 0.00972052400000000469
```

```
In [141]: time 500 (test_boolarray 500 100 110);;

    Temps en secondes: 0.008604s
```

```
Out[141]: - : float = 0.00860422200000000685
```

```
In [142]: time 500 (test_Int32Set 500 100 110);;

    Temps en secondes: 0.024973s
```

```
Out[142]: - : float = 0.024972699999999997
```

2.5 Conclusion

Tout ça n'a pas servi à grand chose, on a réussi à montrer que pour des petits entiers, utiliser un `bool array` de taille 32 (fixe) est plus efficace qu'utiliser ces `bit sets`.

Ça suffit pour aujourd'hui !

Allez voir d'autres notebooks que j'ai écrit.