Pourquoi faire du C quand on peut faire pire en OCaml

Ou comment minimiser les allocations, quand c’est possible, mais pas forcement souhaitable.

Pourquoi essayer de faire ça ?

- Pour aller plus vite ?
- Pour réduire la variance du temps d’exécution ?
- Parce que ?
- Pour faire des choses trèèèèès sales !

Mais ce n’est pas ça le sujet aujourd’hui !

Demo

base : 388

-inline plein : 352

flambda : 383 (bof…)

flambda -O3 : 206

flambda -O3 -inline-call-cost plein : 181

flambda2 -pareil : 233 (Hum?)

Bien mais peut faire mieux !

```
- type vec = { x : float; y : float }
+ type vec = { mutable x : float; mutable y : float }

-181
+289
```

Ne faites pas ça à la maison, cette démonstration vous est faite par des professionnels avec des années de pratique !

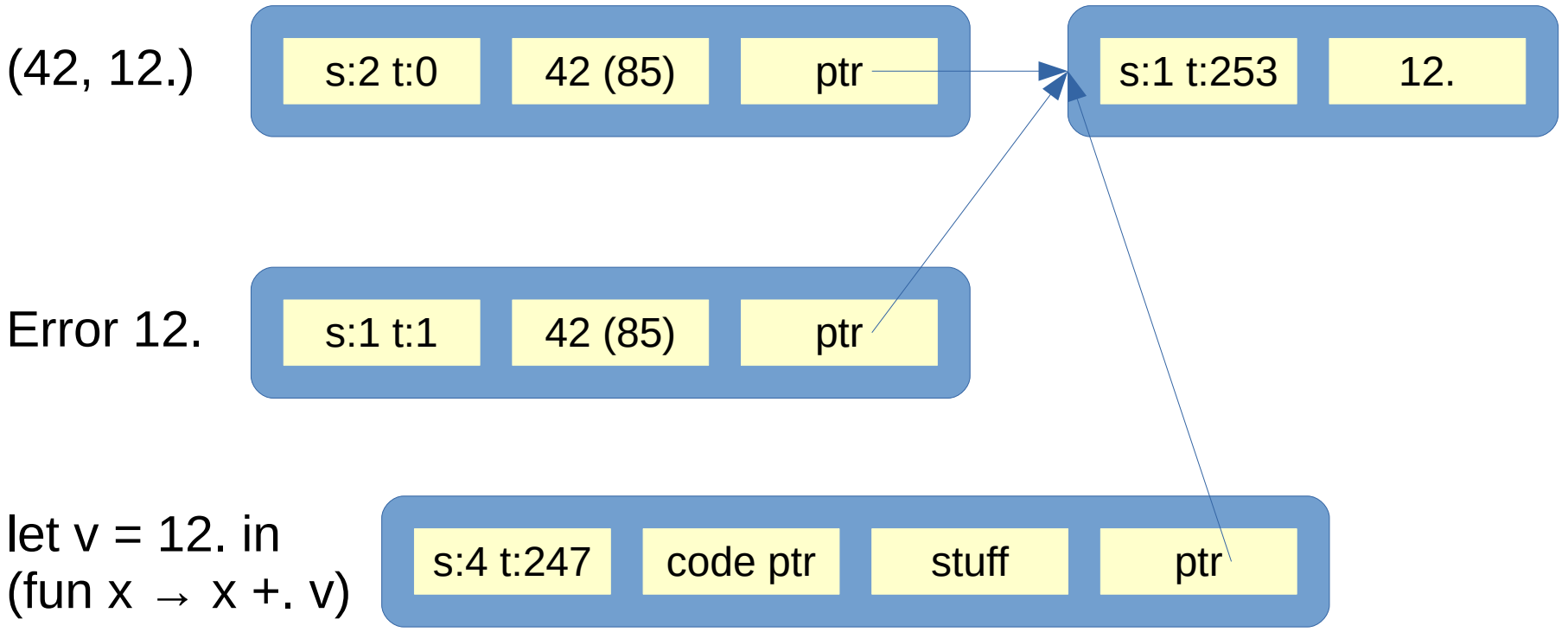
(Sur le compilateur ou Tezos par exemples, pour ne citer que des exemples ou la qualité de code compte!)

Petit rappel du GC OCaml

Il y a 2 types de valeurs :

- Les entiers 31/63 bits (int, char, bool, enums)
- Les blocs (le reste : float, tuples, fonctions …)

Le bit de signe sert à distinguer


(42, 12.)
s:2 t:0
42 (85)
ptr
s:1 t:253
12.
Error 12.
s:1 t:1
42 (85)
ptr
let v = 12. in
(fun x → x +. v)
s:4 t:247
code ptr
stuff
ptr

Constantes

On alloue statiquement les valeurs qui sont toujours vivante et immutables

Très dépendant du compilateur

Constantes

```
let n = 12.
let cpl = (42, n)
let s = Some n
let l = [ 1; 2; 3 ]
let f x = List.map (fun y -> x + y) l


let n = [|1;2;3|]
let v = ref 12.


let a = 11. +. 1.
let f x = 11. +. x
let b = f 1.
```

Constantes (flambda)

```
let b =
  let f x = 11. +. x in
  f 1.
module S = (Set.Make[@inlined])(Int)
let l = (List.map[@unrolled 4]) succ [1;2;3]
```

Ça c’est encore OK, pas de trucs sales pour l’instant

Petite digression sur [@inline]

```
let[@inline] f x = x + 1
let f x = x + 1 [@@inline]

let g x = (f[@inlined]) x

let g x = (f[@inlined always]) x
let g x = (f[@inlined never]) x

module type X = sig end
module Func (X : X) = struct end[@@inline]

module M = (Func[@inlined]) (struct end)
```

Petite digression sur [@unroll]

```
let n = (List.map[@unrolled 4]) succ [1;2;3;4]

let rec list_map f l =
  match l with
  | [] -> []
  | h :: t ->
    f h :: (list_map[@unrolled 1]) f t

let list_succ l = (List.map[@specialised]) succ l
```

Déboîtage (Unboxing)

Pour certaines valeurs locales, on évite l’allocation :
Si elle n’a pas besoin d’être dans le tas, ne pas l’y mettre

```
let sqrti x =
  let x’ = float x in
  let sq’ = sqrt x’ in
  int_of_float sq’
```

```
(function camlSq__sqrti_box_8 (x/450: val)
 (+
   (<<
     (intoffloat
       (extcall "sqrt"
         (load float64u
           (opaque (alloc 1277 (floatofint (>>s x/450 1)))))))
     1)
   1))

(function camlSq__sqrti_21 (x/446: val)
 (+
   (<<
     (intoffloat
       (extcall "sqrt" (floatofint (>>s x/446 1))))
     1)
   1))
```

Remarques

- Pas pour toutes les valeurs
- Problèmes d’heuristique dans flambda 1

```
let swap c = let (a, b) = c in (b, a) (* Yurk *)
let f b x y =
  let c = (x, y) in
  let c' = if b then c else swap c in
  let a, b = c' in
  a - b
```

```
(let
    (f/86
       (closure
         (fun camlDeboite_fl2__f_86:int 3  b/88 x/89[int] y/90[int]
           (let
             (c/91 (makeblock 0 (int,int) x/89 y/90)
              c'/92
                (if b/88 c/91 (makeblock 0 (field 1 c/91) (field 0 c/91))))
             (- (field 0 c'/92) (field 1 c'/92)))) ))
```

```
  let swap c = let (a, b) = c in (b, a) (* Yurk *)
  let f b x y =
    let c = (x, y) in
    let c' = if b then c else swap c in
    let a, b = c' in
    a - b
```

```
(let
    (f/86
       (closure
         (fun camlDeboite_cl__f_86:int 3  b/88 x/89[int] y/90[int]
           (let
             (c/91 (makeblock 0 (int,int) x/89 y/90)
              c'/92
                (if b/88 c/91 (makeblock 0 (field 1 c/91) (field 0 c/91))))
             (- (field 0 c'/92) (field 1 c'/92)))) ))



(function camlDeboite_fl2__f_1_3_code
     (b102/295: val x103/296: val y104/297: val)
 (catch
   (if (== 1 b102/295) (exit 8 x103/296 y104/297) (exit 8 y104/297 x103/296))
 with(8 unboxed_field_1105/298: val unboxed_field_0106/299: val)
   (+ (- unboxed_field_0106/299 unboxed_field_1105/298) 1)))
```

Avec flambda2

```
let g b x =
  let o = if b then Some x else None in
  let n = match o with | None -> 1 | Some n -> n in
  x + n
```

```
(function camlDeboite_fl2__g_0_1_code
    (b66/279: val x67/280: val)
 (catch (if (> b66/279 3) (exit 3 0) (exit 3 1)) with(3 is_int68/281: int)
   (catch (if is_int68/281 (exit 2 3) (exit 2 x67/280)) with(2 n69/282: val)
     (+ (+ x67/280 n69/282) -1))))
```

Certaines cas sont toujours boîtés

- Arguments de fonction*
- Retour de fonction*
- Dans une valeur sur le tas

*Sauf quand ce n’est pas le cas

Pour les arguments

1. Ne pas avoir d’arguments !
2. Pas d’arguments structurés
3. Attendre une optimisation magique ?
4. Ne pas avoir de fonctions !

Pas d’arguments structuré

```
type t = { x : int ; y : int }

let scalaire a b =
  a.x * b.x + a.y * b.y
```

```
let scalaire ax ay bx by =
  ax * bx + ay * by
```

Pas d’arguments structuré

```
type t = Int of int | Float of float

let plus_one t =
  match t with
  | Int i -> Int (i + 1)
  | Float f -> Float (f +. 1.)
```

```
type _ t = Int : int t | Float : float t

let plus_one (type x) (t : x t)
  (n : x) : x =
  match t with
  | Int -> n + 1
  | Float -> n +. 1.
```

Bien sûr ça ne marche pas avec une liste

Optimisation magique ?

Faisable : J’avait un vieux proto (avant flambda) pour faire ça.

Prévu pour la seconde version de flambda 2 (2.1?)

Ne pas avoir de fonction

L’inlining est là pour nous sauver !

```
type t = { x : int ; y : int }

let scalaire a b =
   a.x * b.x + a.y * b.y
[@@inline always]
```

Pour les résultats

1. Ne pas avoir de résultat !
2. Ne pas avoir de retour de fonction !
3. Pas de résultats structurés
4. Attendre une optimisation magique ?
5. Ne pas avoir de fonctions !

Pas de retour de fonctions ?

Ne faire que des appels terminaux !

```
type t = { x : int; y : int }
let plus a b = { x  = a.x * b.x; y = a.y * b.y }

let plus_k a b k =
  k { x  = a.x * b.x; y = a.y * b.y }

let plus_k_bien a_x b_x k =
  k ~x:(a_x * b_x) ~y:(a_y * b_y)
```

Faire attention au capture dans les fermetures
Ne marche pas pour les boucles

Pas de retour de fonctions !

On a une bonne source d’inspiration pour où aller piocher !

Les langages à expression c’est mignon, mais les langages à statements* c’est tellement mieux non ?

*Une bonne traduction de statement ? Déclaration ? Prédicat ?

Des variantes

```
type t = { mutable x : int; mutable y : int }

let retour_plus = { x = 0; y = 0 }
let plus a b =
  retour_plus.x <- a.x * b.x;
  retour_plus.y <- a.y * b.y

let plus_in_place a b =
  a.x <- a.x * b.x;
  a.y <- a.y * b.y
```

Inspirons nous jusqu'au bout !

```
let gros_bytes = ref (Bytes.create gros_nombre)
let malloc (taille:int) : int = ...

type t = { x : int; y : int }
let plus_t a b = { x  = a.x * b.x; y = a.y * b.y }

let lire_t p =
  { x = Int64.to_int (Bytes.get_int64_ne !gros_bytes p);
    y = Int64.to_int (Bytes.get_int64_ne !gros_bytes (p + 8)); }
let ecrire_t p t =
  Bytes.set_int64 !gros_bytes p (Int64.of_int t.x);
  Bytes.set_int64 !gros_bytes (p+8) (Int64.of_int t.x)
let plus p1 p2 d = ecrire_t d (plus_t (lire_t p1) (lire_t p2))
```

Références locales

Si une référence ne échappe clairement pas, alors elle peut ne pas exister.

(Les références ont une vilaine tendance à vouloir fuir)

Références locales

```
type res = { mutable res : float }
let sum a =
  let acc = ref 0. in
  for i = 0 to Floatarray.length a - 1 do
    acc := Floatarray.get a i +. !acc
  done;
  res := !acc
```

Array.fold ou Array.iter marchent si inlinés (avec flambda)

Conclusion

- Faut il toujours écrire du code comme ça ?
  Probablement pas. Sauf si vous êtes JaneStreet ?
- Mais c’est acceptable localement dans des cas particuliers
- Ce n’est pas garanti d’être plus rapide.
  Fait naïvement c’est souvent plus lent.
  Le GC mineur de OCaml est terriblement efficace.
  Et le majeur a une latence très faible.
- Les allocations qui comptent vraiment sont celles du tas majeur.
  les autres sont le travail du compilateur.
- Dans tous les cas il faut passer au banc d’essai (benchmarker...)

Resultat sur la démo ?

- Allocations : 181 => 0
- Temps : 0.75s => 0.39s

Bonus : inliner trick

```
let inliner f x = (f[@inlined]) x [@@inline]
let noinliner f x = f x [@@inline]

let generic_code x =
  if cond x then usefull_to_inline
  else usefull_to_inline

let truc inliner g x =
  inliner generic_code x;
  inliner generic_code x;
  inliner generic_code x;
  inliner generic_code x
```