Profiling and Optimizing OCaml Programs

OCaml PRO – Grégoire Henry

March 9-13 2015

Outline

An optimising compiler
Manual optimisation
Debugging & profiling

An optimising compiler

ocamlc moto: keep the compiler code simple!

ocamlopt an *optimising* compiler:

- not usual ”imperative” optimisations;
- but specialised to idiomatic OCaml code.

Before to profile and to optimise, we should:

- understand the optimisation automatically introduced by the compiler;
- be able to read and understand the successive intermediate languages.

AST & typed AST no optimisations, option `-dparsetree` or `-dtypedtree`;

lambda-code option `-dlambda`:

- pattern-matching compilation
- local reference unboxing

lambda-code with explicit closure option `-dclambda`:

- small function inlining
- constant propagation

cmm options option `-dcmm`:

- arithmetic operations simplification
- local float, in32, int64 unboxing
- local integer untagging

mach, linear, assembly more classical optimisations, option `-S`:

- register allocation with coalescing
- common sub-expression elimination
- allocation simplification

Pattern-matching

```
1 : let f = function
2 : | [] -> 0
3 : | 1 :: 3 :: _ -> 1
4 : | 1 :: 2 :: _ -> 3
5 : | 2 :: _ -> 4
6 : | _ -> 5
```

sum types The compiler generate the minimal number test to discriminate value constructors. It uses table jump when efficient.

string optimal matching (since 4.02.0). Word by word, most discrimininating word first.

conditional clauses reduce optimisations

Lambda-code:

```
1 : (function param/1017
2 :   (catch
3 :     (if param/1017
4 :       (let (match/1018 =a (field 0 param/1017))
5 :         (if (!= match/1018 1) (if (!= match/1018 2) (exit 1) 4)
6 :           (let (match/1019 =a (field 1 param/1017))
7 :             (if match/1019
8 :               (let (match/1020 =a (field 0 match/1019))
9 :                 (if (!= match/1020 2) (if (!= match/1020 3) (exit 1) 1)
10 :                   3))
11 :               (exit 1)))))
12 :       0)
13 :    with (1) 5)))
```

Local references

Local reference are unboxed. For instance: cpt is unboxed, cpt2 is not.

```
1 : let sum a =
2 :   let cpt = ref 0 and cpt2 = ref 0 in
3 :   for i = 0 to Array.length a - 1 do
4 :     cpt := !cpt + a.(i)
5 :   done;
6 :   Array.iter (fun x -> cpt2 := !cpt2 + x) a;
7 :   !cpt, !cpt2
```

Lambda-code:

```
1 : (function a/1063
2 :   (let (cpt/1064 =v 0 cpt2/1065 = (makemutable 0 0))
3 :     (seq
4 :       (for i/1066 0 to (- (array.length a/1063) 1)
5 :         (assign cpt/1064 (+ cpt/1064 (array.get a/1063 i/1066))))
6 :       (apply (field 11 (global Array!))
7 :         (function x/1067
8 :           (setfield_imm 0 cpt2/1065 (+ (field 0 cpt2/1065) x/1067)))
9 :         a/1063)
10 :      (makeblock 0 cpt/1064 (field 0 cpt2/1065)))))
```

Scalar unboxing

Short lived scalar value that do not escape are not boxed.

```
1 : let f x y =
2 :   let a = x +. y in
3 :   let b = a *. x in
4 :   let c = b *. y in
5 :   let d = b +. x in
6 :   let e = a +. y in
7 :   (b, e)
```

Cmm code:

```
1 : (function caml10-optimisations__f_1008 (x/1009: addr y/1010: addr)
2 :  (let
3 :    (a/1018 (+f (load float64u x/1009) (load float64u y/1010))
4 :     b/1019 (*f a/1018 (load float64u x/1009)) b/1012 (alloc 1277 b/1019)
5 :     c/1020 (*f b/1019 (load float64u y/1010))
6 :     d/1021 (+f b/1019 (load float64u x/1009))
7 :     e/1022 (+f a/1018 (load float64u y/1010)) e/1015 (alloc 1277 e/1022))
8 :     (alloc 2048 b/1012 e/1015)))
```

Remarks: $1277 = 1 \ll 10 + 253$ is the immediate value representing the header of block of tag 253 (`Double_tag`) and size 1. And $2048 = 2 \ll 10$ is the header of a block of tag 0 and size 2.

Allocation simplification

In a late compilation phase, successive allocation are grouped:

```
1 : let f x y z =
2 :   let tail = y :: z in
3 :   let list = x :: tail in
4 :   let record = { list = (x,y); len = 3 } in
5 :   (list, record, ref (x +. y))
```

Mach code (-dcombine):

```
1 :   ...
2 :   tail/32 := alloc 160
3 :   [tail/32 + -8] := 2048 (init)
4 :   [tail/32] := y/30 (init)
5 :   [tail/32 + 8] := z/31 (init)
6 :   list/33 := tail/32 + 24
7 :   [list/33 + -8] := 2048 (init)
8 :   [list/33] := x/29 (init)
9 :   [list/33 + 8] := tail/32 (init)
10 :  ...
```

Register allocation

- Function arguments are passed on registers, so is the return argument.
- Local value are stored on registers when possible:
  - register are reused when a local value is not used anymore;
  - stack is used when missing registers.
- Before any call to the GC or any function call, registers are backuped on the stack, i.e. all registers are *caller-save*. They are *GC roots* and may reference live blocks.
  - except for external function call with the `"noalloc"` flag, where the *callee-save* registers, of the C ABI, are not backuped.

Until the current version of OCaml, the inlining decision only depends on the function definition: i.e. a given function is always inlined, or never.

- The main criteria is the function size. The threshold may be modified with:

```
-inline N
```

- Any function containing a constant string or a local function may never be inlined.

Inproper inlining may decrease performances (code explosion). Use the option `-inline` only on modules with function that may be simplified.

Hints: how to known if a function will be inlined?

In the following compilation unit `module.cmx`:

```
1: let f x y z = x+.y+.z
2: let g l = List.map (fun x -> x) l
```

- the function `f` is inlined,
- while the function `g` is not.

```
# ocamlobjinfo module.cmx
[...]
Approximation:
  (0: function camlModule__f_1008 arity 3 (closed) (inline) ->  _;
   1: function camlModule__g_1012 arity 1 (closed) ->  _)
[...]
```

Manual optimisation

You know that!

Usual warnings:

- ensure correction before doing performance analysis and optmisation;
- on non-critical code path, prefer clarity to optimality.

In particular for OCaml:

- initially, use the simplest and quickest pattern to write;
- check the asymptotic complexity of your algorithm;
- do not care about memory management until at least 20% of CPU time is spent on garbage collection.

The standard library contains very efficient data structures:

- Persistent structures
  - List: contains all generic iterators
  - Map, Set: balanced binary tree
- Imperative structures
  - Hashtbl: based on MurmurHash 3
  - Queue, Stack
  - Buffer

Manual optimisation: outline

- Tail-rec transformation
- Polymorphism elimination
- Introduction of static closure
- Deforestation
- Defonctorisation
- Early interuption in loop
- Data structure specialisation

Tail-rec

- We have done that already!
- Don't bother until you really need it!
  - You have been hit by a stack overflow;
  - The function is on your critical path.
- Recursion is usually faster than a `while` loop.

Polymorphism elimination

The compiler has a few type-directed optimisation that are disabled by the polymorhism.

Comparison polymorphic comparison is an external call, while monomorphic comparison may be compiled to a specialised code.

```
1: let f x y = x < y
2: let g (x: int) (y: int) = x < y
3: let h (x: float) (y: float) = x < y
```

```
1: (let
2:   (f/1008 = (function x/1009 y/1010 (caml_lessthan x/1009 y/1010))
3:    g/1011 = (function x/1012 y/1013 (< x/1012 y/1013))
4:    h/1014 = (function x/1015 y/1016 (<. x/1015 y/1016)))
5:   (makeblock 0 f/1008 g/1011 h/1014))
```

Array access With polymorphic arrays the compiler has to dynamically check whether the underlying representation is a generic array or an float array.

Static allocation of closure

Non-closed local definition of a function requires dynamic allocations:

```
1 : let insert x xs =
2 :   let rec loop = function
3 :     | [] -> [x]
4 :     | y :: _ as xs when x < y -> x :: xs
5 :     | y :: _ as xs when x = y -> xs
6 :     | y :: xs -> y :: x :: loop xs in
7 :   loop xs
```

Closed local definition are statically allocated:

```
1 : let insert x xs =
2 :   let rec loop x = function
3 :     | [] -> [x]
4 :     | y :: _ as xs when x < y -> x :: xs
5 :     | y :: _ as xs when x = y -> xs
6 :     | y :: xs -> y :: x :: loop x xs in
7 :   loop xs
```

Global values do not count in the set of free variables.

Deforestation

While usually more readable, "overly functional" code may allocate too many temporary values:

```
1 : let sum_squares list =
2 :   let squares = List.map (fun x -> x*x ) list in
3 :   let sum = List.fold_left (+) 0 squares in
4 :   sum
```

```
1 : let sum_squares list =
2 :   List.fold_left (fun acc x -> acc + x*x) 0 squares
```

Defunctorisation

In ocaml-4.02, functorised code is generic code, it is usually less efficient:

- (even) less type-directed optimisations
- no static allocation of closures

```
1 : module IntMap =
2 :   Map.Make(struct
3 :     type t = int
4 :     let compare = compare
5 :   end)
```

Note: this should be fixed in ocaml-4.03, with a complete rewriting of the inliner.

A bad pattern:

```
1 : let first_positive list =
2 :   let first = ref None in
3 :   List.iter
4 :     (fun x -> if x > 0 then
5 :        match !first with
6 :        | None -> first := Some x
7 :        | Some _ -> ())
8 :     list;
9 :   match !first with
10 :  | None -> raise Not_found
11 :  | Some x -> x
```

Another bad pattern:

```
1 : let first_positive xs =
2 :   let first =
3 :     List.fold_left
4 :       (fun acc x ->
5 :          match acc with
6 :          | None when x > 0 -> Some x
7 :          | _ -> acc)
8 :       None
9 :       xs in
10 :  match first with
11 :  | None -> raise Not_found
12 :  | Some x -> x
```

More efficient patterns:

- Use recursion:

```
1 : let rec first_positive = function
2 :   | [] -> raise Not_found
3 :   | x :: _ when x > 0 -> x
4 :   | _ :: xs -> first_positive xs
```

- Use (local) exceptions:

```
1 : let first_positive xs =
2 :   let module M = struct exception E of int end in
3 :   try
4 :     List.iter (fun x -> if x > 0 then raise (M.E x)) xs;
5 :   with M.E x -> x
```

Data structure specialisation

- Use the most efficient data structures for the current task!
- If no data structure is efficient enough for all task, convert. For instance:
  - Read lines of file in a list is more efficient;
  - Sorting an array is more efficient that sorting a list.

```
1 : let lines : string list = read_lines "a_file";;
2 : let sorted_lines : string array =
3 :   Array.sort compare (Array.of_list lines);;
4 : let () = write_lines sorted_lines "a_sorted_file"
```

Debugging & profiling

Debugging

- compile with `-g`
- ocamldebug (bytecode only)
  - break point; strack introspection;... ; and
  - if allows to step back!
- gdb might helps (rarely)
  - ongoing work: let the compiler export enough DWARF code (JaneStreet)
- sadly: tracing with printf!
  - use `ppx_deriving` to generate printers
- set `OCAMLRUNPARAM=b` to display backtrace for uncaught exceptions

Debugging memory

Warning: if there is less than 20% of CPU time in GC, there is probably not much to be optimised.

Automatic garbage collection does not prevents from all form a memory leaks! For instances:

- over-allocation;
- unused references to a containers.

There is not (yet) a lot of memory debugging tools.

- Manual tracing: `Gc.quick_stat` and `Gc.stat`.
- Visual tracing: ocp-memprof.
- ...

Profiling

- `ocamlprof`
- `perf` (on linux)
  - better result when using the `-with-frame-pointer` when compiling the OCaml compiler
  - `perf record -g ./my_prog`
  - `perf report -g`