High level OCaml optimisations

Pierre Chambart, OCamlPro

OCaml 2013, 23 September 2013

OCaml is fast

Not an optimising compiler, but:

- Predictable performances
- Good generated code

What can we do to get faster ?

Small modification on high level shouldn't influence too much low level

```
let f x =
  let cmp = x > 3 in
  if cmp then A
  else B

let g x =
  if x > 3 then A
  else B
```

- which one is faster ?
- g is faster: peephole

Abstract code could be compiled less abstractly

```
let g x =
  let f v =
    x + v
  in
  f 3
```

- f inlined, but its closure allocated at each call.
- But we don't want the compiler to be 'too smart'.

How it works

```
parsetree typedtree lambda clambda cmm mach lin asm
                            '-> byte code
```

- parse tree: AST
- typed tree: AST with types
- lambda: untyped lambda
- clambda: lambda + closures
- cmm: simple C-like
- mach: instruction graph (llvm-like)
- lin: almost like assembly

How it works

```
parsetree typedtree lambda clambda cmm mach lin asm
                            '-> byte code
```

- typedtree to lambda: high level construct elimination
- lambda: high level simplifications
- lambda to clambda: closure introduction, inlining, constant propagation (and book keeping)
- clambda to cmm: unboxing, lots of peep hole
- cmm to mach: instruction selection
- mach: allocation fusion, register allocation, scheduling

Closures

```
let g x =
  let f v =
    x + v
  in
  f 3
```

```
let g x =
  let closure_f = { x = x } in
  let f v closure_f =
    closure_f.x + v
  in
  f 3 closure_f
```

Where:

- typedtree: too complicated
- lambda: we want inlining, simpler with closures
- clambda: difficult to improve (I tried)
- cmm: good for local optimisation
- mach: architecture specific

Between lambda and clambda

```
parsetree -> typedtree -> lambda -> flambda -> clambda -> cmm -> mach -> lin -> asm
```

We need:

- High level
- Simple manipulation
- Explicit closures
- Explicit value dependencies

flambda: lambda + explicit symbolic closures (normal and Administrative Normal Form)

Difference with clambda

```
let g x =
  let closure_f = { x = x } in
  let f v closure_f =
    closure_f.x + v
  in
  f 3 closure_f
```

```
let g x =
  let closure_f = [|code_pointer; 3; x|] in
  let f v closure_f =
    closure_f.(2) + v
  in
  f 3 closure_f
```

New transformations

- lambda to flambda: closure introduction.
- flambda to clambda: mainly book-keeping (and preparing cross module informations)

The magic will be in flambda to flambda passes.

Optimisation framework

Transformations provided to simplify passes:

Input: cannonical representation Few restrictions on output.

- inlining
- dead code elimination
- constant propagation/simplification

Not optimising: simplification to allow good code generation

Constant extractions

```
let a = (1,2)

let f x =
  let y = (a,3) in
  x, y
```

```
let a = (1,2)
let y = (a,3) in

let f x =
  x, y
```

Inlining

```
let g x =
  let closure_f = { x = x } in
  let f v closure_f =
    closure_f.x + v
  in

  f 3 closure_f
```

```
let g x =
  let closure_f = { x = x } in
  let f v closure_f =
    closure_f.x + v
  in

  let v = 3 in
  closure_f.x + v
```

Simplification

```
let g x =
  let closure_f = { x = x } in
  let f v closure_f =
    closure_f.x + v
  in
  let v = 3 in

  closure_f.x + v
```

```
let g x =
  let closure_f = { x = x } in
  let f v closure_f =
    closure_f.x + v
  in
  let v = 3 in

  x + 3
```

Dead code elimination

```
let g x =

  let closure_f = { x = x } in
  let f v closure_f =
    closure_f.x + v
  in
  let v = 3 in

  x + 3
```

```
let g x =

  x + 3
```

Simple optimisation: Lambda lifting

```
let g x =
  let f v =
    x + v
  in
  f 3
```

Simple optimisation: Lambda lifting

```
let g x =
  let f v =
    x + v
  in
  f 3
```

```
let g x =
  let f' x v =
    x + v
  in
  let f v = f' x v in
  f 3
```

- ~20 lines
- No need to bother propagating: it's the inliner's job.

```
let g x =
  let f' x v =
    x + v
  in
  f' x 3
```

Change the performance model:

- Now: WYSIWYG
- Wanted: Some kind of understandable compile time evaluation

```
let map f l =
  let rec aux = function
    | [] -> []
    | h::t -> f h :: aux t
  in
  aux l

let f l = map succ l
```

Future

- High level things in cmm could move to flambda
- Lots of small simple passes

One last thing

- Please add build_test to your opam packages !
- No Obj.{magic, set_field} or whatever horrible thing: I will break your code !

Flambda type

```
type 'a flambda =
  | Fclosure of 'a ffunctions * 'a flambda IdentMap.t * 'a
  | Foffset of 'a flambda * offset * 'a
  | Fenv_field of 'a fenv_field * 'a

  | Fsymbol of symbol * 'a
  | Fvar of Ident.t * 'a
  | Fconst of const * 'a
  | Fapply of 'a flambda * 'a flambda list * offset option * Debuginfo.t * 'a

  | Flet of let_kind * Ident.t * 'a flambda * 'a flambda * 'a
  | ...

  | Funreachable of 'a

and const =
  | Fconst_base of constant
  | Fconst_pointer of int
  | Fconst_float_array of string list
  | Fconst_immstring of string
```

Numbers

- knuth-bendix ~20%
- noiz ~40%
- set ~20%

Knuth-bendix

```
let f x = if x = 0 then failwith "error"
```

compiled as

```
let exn = Failure "error"
let f x = if x = 0 then raise exn
```

inlining

- noiz `ocaml let map_triple f (a,b,c) = (f a, f b, f c)`
- set: functor