Optimisations you shouldn't do

MAY
24
2013

Posted by Pierre Chambart - 0 Comments

Doing the compiler's work

Working at OCamlPro may have some drawbacks. I spend a lot of time hacking the OCaml compiler. Hence when I write some code, I have a good glimpse of what the generated assembly will look like. This is nice when I want to write performance sensitive code, but as I usually write code for which execution time doesn't matter much, this mainly tends to torture me. A small voice in my head is telling me "you shouldn't write it like that, you known you could avoid this allocation". And usually, following that indication would only tend to make the code less readable. But there is a solution to calm that voice: making the compiler smarter than me.

OCaml compilation mechanisms are quite predictable. There is no dark magic to replace your ugly code by a well-behaving one, but it always generates reasonably efficient code. This is a good thing in general, as you won't be surprised by code running more slowly than what you usually expect. But it does not behave very well with dumb code. This may not often seem like a problem with code written by humans, but generated code, for example coming from camlp4/ppx, or compiled from another language to OCaml, may fall into that category. In fact, there is another common source for non-human written code: inlining.

Inlining

Inlining (or inline expansion) is a key optimisation in many compilers and particularly in functional languages. Inlining replaces a function call by the function body. Let's apply inlining to f in this example.

```
let f x = x + 1
let g y = f (f y)
```

We replace the calls to f by let for each arguments and then copy the body of f.

```
let g y =
  let x1 = y in
  let r1 = x1 + 1 in
  let x2 = r1 in
  let r2 = x2 + 1 in
  r2
```

Inlining allows to eliminate the cost of a call (and associated spilling), but the main point is elsewhere: it puts the code in its context, allowing its specialisation. When you look at that generated code after inlining your trained eyes will notice that it looks quite dumb. And you really want to rewrite it as:

```
let g y = y + 2
```

The problem is that OCaml is compiling smart code into smart assembly, but after inlining your code is not as smart as it used to be. What is missing in the current compiler is a way to get back a nice and smart code after inlining. (To be honest, OCaml is not that bad and on that example it would generate the right code: put this on the sake of the mandatory blog-post dramatic effect.)

In fact you could consider inlining as two separate things: duplication and call elimination. By duplication you make a new version of the function that is specialisable in its context, and by call elimination you replace the call by specialised code. This distinction is important because there are some cases where you only want to do duplication: recursive functions.

Recursive function inlining

In a recursive function duplicating and removing a call is similar to loop unrolling. This can be effective in some cases, but this is not what we want to do in general. Lets try it on `List.map`

```
let rec list_map f l = match l with
  | [] -> []
  | a::r -> f a :: list_map f r

let l' =
  let succ = (fun x -> x + 1) in
  list_map succ l
```

If we simply inline the body of list_map we obtain this

```
let l' =
  let succ = (fun x -> x + 1) in
  match l with
  | [] -> []
  | a::r -> succ a :: list_map succ r
```

And with some more inlining we get this which is probably not any faster than the original code.

```
let l' =
  let succ = (fun x -> x + 1) in
  match l with
  | [] -> []
  | a::r -> a + 1 :: list_map succ r
```

Instead we want the function to be duplicated.

```
let l' =
  let succ = (fun x -> x + 1) in
```

OCaml Cheat Sheets

May 2011

OCaml 32bits longval

```
  let rec list_map' f l = match l with
    | [] -> []
    | a::r -> f a :: list_map' f r in
  list_map' succ l
```

Now we know that `list_map'` will never escape its context hence that its f parameter will always be succ. Hence we can replace `f` by succ everywhere in its body.

```
let l' =
  let succ = (fun x -> x + 1) in
  let rec list_map' f l = match l with
    | [] -> []
    | a::r -> succ a :: list_map' succ r in
  list_map' succ l
```

And we can now see that the `f` parameter is not used anymore, we can eliminate it.

```
let l' =
  let succ = (fun x -> x + 1) in
  let rec list_map' l = match l with
    | [] -> []
    | a::r -> succ a :: list_map' r in
  list_map' l
```

With some more inlining and cleaning we finally obtain this nicely specialised function which will be faster than the original.

```
let l' =
  let rec list_map' l = match l with
    | [] -> []
    | a::r -> a + 1 :: list_map' r in
  list_map' l
```

Current state of the OCaml inliner

Inlining can gain a lot, but abusing it will increase code size a lot. This is not only a problem of binary size (who cares?): if your code does not fit in processor cache anymore, its speed will be limited by memory bandwidth.

To avoid that, OCaml has a threshold to the function size allowed for inlining. The compiler may also refuse to inline in other cases that are not completely justified though, mainly for reasons related to its architecture:

- duplication and call elimination are not separated, hence recursive function duplication is not possible.
- functions containing structured constants or local functions are not allowed to be duplicated, preventing those functions to be inlined.

```
let constant x =
  let l = [1] in
  x::l

let local_function x =
  let g x = some closed function in
  ... g x ...
```

The assumption is that if a function contains a constant or a function it will be too big to be reasonably inlined. But there is a reasonable transformation that could allow it.

```
let l = [1]
let constant x =
  x::l

let g x = some closed function
let local_function x = ... g x ...
```

and then we can reasonably inline `constant` and `local_function`. Those cases are only technical

limitation that could easily be lifted with the new implementation.

But improving the OCaml inliner is not that easy. It is well written, but it is also doing a lot of other things at the same time:

closure conversion

closure conversion transforms functions to a data structure containing a code pointer and the free variables of the function. You could imagine it as that transformation:

```
let a = 1

let f x = x + a (* a is a free variable in f *)

let r = f 42
```

Here `a` is a free variable of `f`. We cannot compile `f` while it contains reference to free variables. To get rid of the free variables, we add a new parameter to the function, the environment, containing all the free variables.

```
let a = 1

let f x environment =
  (* the new environment parameter contains all the free variables of f *)
  x + environment.a

let f_closure = { code = f; environment = { a = a } }

let r = f_closure.code 42 f_closure.environment
```

Value analysis

In functional languages inlining is not as simple as it is for languages like C because the function name does not tells you which function is used at a call site:

```
let f x = (x,(fun y -> y+1))

let g x =
  let (a,h) = f x in
  h a
```

To be able to inline h as (fun y -> y+1) the compiler needs to track which values flows to variables. This is usually called a value analysis. This example can look a bit contrived, but in practice functor application generate quite similar code. This allows for instance to inline Set.Make(...).is_empty. The result of this value analysis is used by other optimisations:

Constant folding

When the value analysis can determine that the result of an operation is a constant, it can remove its computation:

```
let f x =
  let a = 1 in
  let b = a + 1 in
  x + b
```

Since `b` always have the value `2` and `a + 1` does not have side effects it is possible to simplify it.

```
let f x =
  x + 2
```

Direct call specialisation

Sometimes it is impossible to know which function will be used at a call site:

```
let f g x = g x
```

There is a common representation (the closure) that allows to call a function without knowing anything about it. Using a function through its closure is called a generic call. This is efficient, but of course not as efficient as a simple assembly call (a direct call). The work of the direct call specialisation is to turn as many as possible generic call into direct ones. In practice, the vast majority of calls can be optimised.

Improving OCaml inliner

The current architecture is very fast and works well on a lot of cases, but it is quite difficult to improve the handling of corner cases.

I have started a complete rewrite of those passes, I am currently working on splitting all those things in their own passes. The first step was to add a new intermediate representation (flambda) more suited to doing the various analysis. The main difference with the current representation (clambda) is that closures are explicitly represented, making them easier to manipulate. As a nice side effect this intermediate representation allows to plug passes in or out, or loop on them without changing anything to the architecture. But we are losing the possibility to enforce some invariants in the type of the representation, hence we need to be careful to correctly maintain them.

With this new architecture, the closure conversion is done first (going from lambda to flambda). Then on flambda are provided a set of simple analysis:

- simple intraprocedural value and alias analysis
- purity analysis
- constant analysis

- dead expression analysis

And there is a set of simple passes using their results:

- dead code elimination
- constant folding/direct call specialisation/type specialisation: a simple traversal replacing expressions with more efficient ones when the result of the value analysis allows it.
- alias rebinding: Use results of alias analysis to know when a field access can be simplified:

```
let f x =
  let tuple = (x,x) in
  let (y,z) = tuple in
  y + z

let f x =
  x + x
```

Of course nobody would write that, but access to variables bounded in a closure can looks a lot like that after inlining:

```
let f x =
  let g y = x + y in
  g x
```

After closure conversion we obtain this.

```
let f x =
  let g_closure =
    { code = fun x environment -> environment.x + y;
      environment = { x = x } } in
  g_closure.code x g_closure.environment
```

And after inlining `g`.

```
let f x =
  let g_closure =
    { code = fun x environment -> environment.x + y;
      environment = { x = x } } in
  x + closure.environment.x
```

Inlining `g` makes some code that looks a bit stupid. `closure.environment.x` is always the same value as `x`. So there is no need to access it through the structure.

```
let f x =
  let g_closure =
    { code = fun x environment -> environment.x + y;
      environment = { x = x } } in
  x + x
```

Now that we have simplified the code, we notice that g_closure is not used anymore, and dead code elimination can simply get rid of it.

```
let f x =
  x + x
```

- a really, really dumb inliner: it inlines almost anything. Its interest is to demonstrate what can be achieved when putting some code in its context.

After the different optimisation passes we need to send the result to the compiler back-end. This is done by the final conversion from flambda to clambda, which is mainly doing a lot of bureaucratic transformations and mark constant structured values. Doing this constant marking separately also allows to improve a bit the generated code.

```
let rec f x =
  let g y = f y in
  g x
```

`f` and `g` are closed functions but the current compiler will not be able to detect it and allocate a closure for g at each call of f.

Hey ! Where are the nice charts ?

As you noticed that there were no fancy improvements charts, and there won't be any below. Those are demonstrations passes, the generated code can (and probably will) be worse than the one generated by the current compiler. This is mainly done to show what can be achieved by combining simple passes and simple analysis and allowing to apply them multiple times. What is needed to get fast code is to change the inlining heuristic (and re-enable cross module inlining).

My current work is to write more serious analysis allowing better optimisations. In particular I expect that a reasonable interprocedural value analysis could help a lot with handling recursive function specialisation.

My future toys

Then I'd like to play a bit with other common things like

- unused parameters elimination: when a function does not use one of its parameters, remove it. This is trivial with simple functions, but it can get a bit tricky with multiply recursive functions. (that kind of code can appear after constant folding with informations from interprocedural analysis)
- lambda lifting: turning closure into closed function by adding arguments. This can eliminate some allocations

```
let f x =
  let g y = x + y in
  g 4
```

If we add the `x` parameter to `g` we can avoid building its closure each time `f` is called.

```
let f x =
  let g y x = x + y in
  g 4 x
```

This can get quite tricky if we want to handle cases like

```
let f x n =
  let g i = i + x in
  Array.init n g
```

We need to add a new parameter to init also to be able to pass it to g.

- common sub-expression elimination:

```
let f x =
  let a = x + 1 in
  let b = x + 1 in
  a + b
```

In `f` We clearly don't need to compute `x + 1` two times

```
let f x =
  let a = x + 1 in
  a + a
```

- earlier unboxing: Floats are boxed in ocaml, this means that there is an indirection when accessing the constant of a value of type float. To reduce the cost of allocating and accessing floats unboxing

eliminates the indirections between some operations. I'd like to try to do this as a flambda pass to be able to use the results of the value analysis.

If you want to play/hack a bit with the demo look at my github branch (be warned, this branch may sometimes be rebased)

Products

OPAM
TypeRex
Project Sponsoring

Services

Technical Support
Consulting
Training

Company

Contact Us
Company
Team
Jobs
Internships