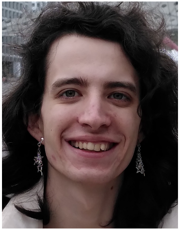
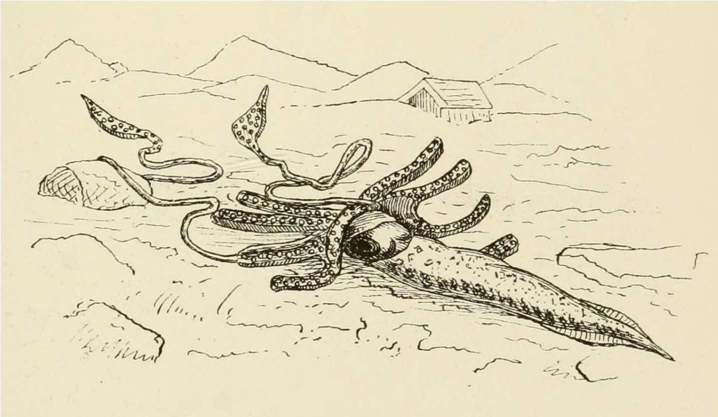
Reducing shapes

Gabriel Scherer, Nathanaëlle Courant

September 18, 2022



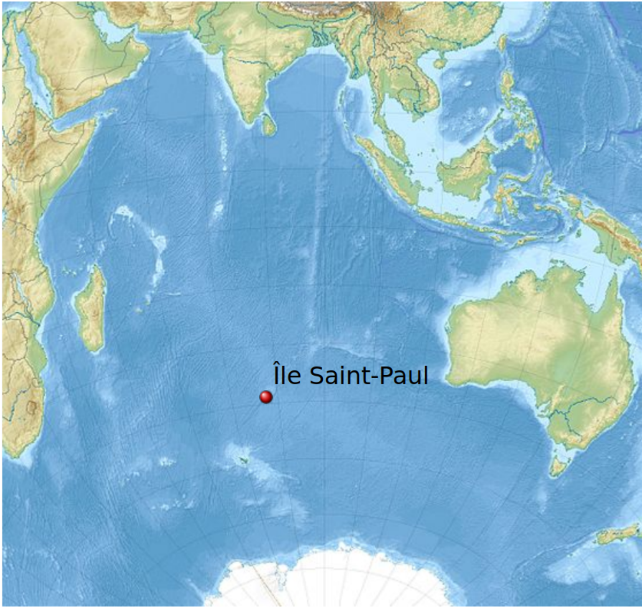
A metaphor



G.W. Tryon Jr. (1879)
https://commons.wikimedia.org/wiki/File:Ommastrephes_mouchezi.jpg
The giant squid that washed ashore on Île Saint-Paul on 2 November 1874.

In case you wonder where Île Saint-Paul is:

In case you wonder where Île Saint-Paul is:



This talk

A field report on the

first sighting (to our knowledge)

of

This talk

A field report on the

first sighting (to our knowledge)

of

strong (by-need) reduction

in the wild, outside proof assistants.

Note: I'm not an expert!

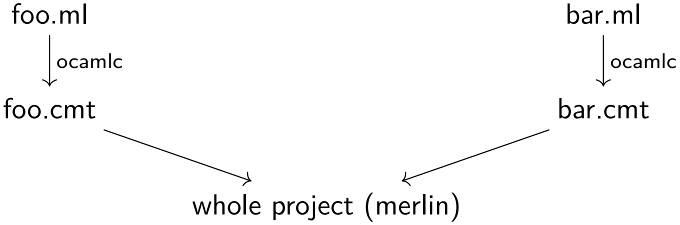
Shapes

Shapes, as designed by Thomas Refis, Ulysse Gérard and Leo White, are λ-terms representing the shape of OCaml modules – and source files. (no term-level information except source locations)

They extended the OCaml compiler to compute shapes and store them in object files.

Motivation: tooling support: "where is `foo` defined?" (requires normalization)

Shape computations



Separate compilation: the shape of a module is an **open term**.

Definition lookup inside functors: we want **strong reduction**.

Problem

A naive implementation of strong reduction does fine in general, but it

Problem

A naive implementation of strong reduction does fine in general, but it

explodes

on some complex functor-using OCaml programs. (Irmin)

Solution

Ulysse Gérard and Thomas Refis implemented some optimizations; enough for "termination" but still unsatisfying.

Strong call-by-need reduction avoids blowups.

Performance on a problematic source file:

	compilation time	output size
no shapes	0.39s	2538Kio : 2.5Mio
shapes, naive +opts	2.15s	91Mio
shapes, strong cbneed	0.40s	2552Kio : 2.5Mio

Why the blowup?

Consider:

module M = **struct**
  let x = A.x
  let y = A.y
  let z = A.z
end

With closed reduction, this only reduces when `A` is a structure/record.

$$\|M\| \leq \|A\|$$

With open reduction, `A` may be neutral: `F(X).Bar`. Then:

$$\|M\| \simeq 3 * \|A\|$$

Actually a very common pattern:

module M = (A : S)

Why the blowup? Intuition

Intuition:
closed, weak reduction has size-exploding examples,
but strong reduction explodes **more**

More precisely:
some realistic closed programs have small normal forms,
but their subterms could blow up under strong reduction.

Thanks!

(Bonus slides follow.)

A terrible implementation

```
let rec eval env : t → t = function
  | Var x →
    Ident.find_same x env
  | Abs (x, t) →
    Abs (x,
     let env' = Ident.add x (Var x) env in
     eval env' t)
  | App (t, u) →
    let f, arg = eval env t, eval env u in
    match f with
    | (Var _ | App _) as ne → App (ne, arg)
    | Abs (x, body) →
      eval (Ident.add x arg env) body
```

A naive implementation (1): types

```
type nf = (* normal forms *)
  | Ne of ne
  | Clos of env * var * t * var * nf
and ne = (* neutral terms *)
  | Var of var
  | App of ne * nf

type open_value =
  | Val of nf
  | Free of var
```

A naive implementation (2): code

```
let rec eval = fun env (t : t) : nf →
  match t with
  | Var x → begin match Ident.find_same x env with
    | Val v → v
    | Free x → Ne (Var x)
    end
  | Abs (x, t) →
    let y = fresh x in
    Clos (env, x, t, y,
     let env' = Ident.add x (Free y) env in
     eval env' t)
  | App (t, u) →
    let f, arg = eval env t, eval env u in
    match f with
    | Ne n → Ne (App (n, arg))
    | Clos (env', x, body, _y, _v) →
      eval (Ident.add x (Val arg) env') body
```

A naive implementation (3): memoization

```
let eval = memo_fix_2 @@ fun eval env (t : t) : nf →
  match t with
  | Var x → begin match Ident.find_same x env with
    | Val v → v
    | Free x → Ne (Var x)
    end
  | Abs (x, t) →
    let y = fresh x in
    Clos (env, x, t, y,
     let env' = Ident.add x (Free y) env in
     eval env' t)
  | App (t, u) →
    let f, arg = eval env t, eval env u in
    match f with
    | Ne n → Ne (App (n, arg))
    | Clos (env', x, body, _y, _v) →
      eval (Ident.add x (Val arg) env') body
```

A by-need implementation (1)

```
type nf = (* normal forms *)
  | Ne of ne
  | Clos of env * var * t * var * dnf
and dnf = nf Lazy.t
and ne = (* neutral terms *)
  | Var of var
  | App of ne * dnf

  let force eval env t = lazy (eval env)
  let delay eval env dv = Lazy.force dv
```

A naive implementation: reminder

```
let eval = memo_fix_2 @@ fun eval env (t : t) : nf →
  match t with
  | Var x → begin match Ident.find_same x env with
    | Val v → v
    | Free x → Ne (Var x)
    end
  | Abs (x, t) →
    let y = fresh x in
    Clos (env, x, t, y,
      let env' = Ident.add x (Free y) env in
      eval env' t)
  | App (t, u) →
    let f, arg = eval env t, eval env u in
    match f with
    | Ne n → Ne (App (n, arg))
    | Clos (env', x, body, _y, _v) →
      eval (Ident.add x (Val arg) env') body
```

A by-need implementation (2)

```
let eval = memo_fix_2 @@ fun eval env (t : t) : nf →
  match t with
  | Var x → begin match Ident.find_same x env with
    | Val v → force eval v
    | Free x → Ne (Var x)
    end
  | Abs (x, t) →
    let y = fresh x in
    Clos (env, x, t, y,
      let env' = Ident.add x (Free y) env in
      delay eval env' t)
  | App (t, u) →
    let f, arg = eval env t, delay eval env u in
    match f with
    | Ne n → Ne (App (n, arg))
    | Clos (env', x, body, _y, _v) →
      eval (Ident.add x (Val arg) env') body
```

A by-need implementation (3)

```
type nf = (* normal forms *)
  | Ne of ne
  | Clos of env * var * t * var * dnf
and dnf = Delayed of env * t
and ne = (* neutral terms *)
  | Var of var
  | App of ne * dnf

  let force eval (env, t) = eval env t
  let delay eval env t = (env, t)
```

A by-need implementation (3)

```
type nf = (* normal forms *)
  | Ne of ne
  | Clos of env * var * t * var * dnf
and dnf = Delayed of env * t
and ne = (* neutral terms *)
  | Var of var
  | App of ne * dnf

  let force eval (env, t) = eval env t
  let delay eval env t = (env, t)
```

If you squint: a by-need version of iterated weak reduction.