Multi-stage programming

Part II: effects and sharing

Jeremy Yallop

International Summer School on Metaprogramming

August 2016

Recap: MetaOCaml

Typed, open

$\Gamma \vdash^{n}$ `e` $: \tau$

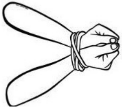
Homogeneous

`.< (* OCaml *) >.`

Generative

Constructors: `.< e >.` `.~e`
Destructors:

Expressiveness



(*Claim*: expressive when used with other features:
polymorphism, algebraic types, modules, overloading, effects, ...)

Recap: aims and approach

Compiler	Program code	Libraries
simple	simple	clever

No abstraction guilt

High-level code transformed for low-level performance

No optimization guilt

Build libraries with reusable domain-specific optimizations

Recap: data representation

Simple (non-Russian) power function:

```
let rec pow : {N:MON} → N.t → int → N.t  =
  fun {N:MON} x n →
    if n = 0 then N.one
    else x * pow x (n - 1)
```

No staging

`t = int`

$2^4 \rightsquigarrow 16$

Possibly-static

`t = Sta of int | Dyn of int code`

$x^4 \rightsquigarrow$ `.< x * x * x * x >.`

Dynamic

`t = int code`

$x^4 \rightsquigarrow$ `.< x * x * x * x * 1 >.`

Partially-static

`t = { sta: int; dyn: int code }`

$2 \times x^4 \times 2 \rightsquigarrow$ `.< 4 * x * x * x * x >.`

Today: $x^4 \rightsquigarrow$ `.< let y = x * x in y * y >.`

Effects + staging

Are effects **helpful**, **harmful**, or **neutral**?

Effects are neutral (w.r.t. staging)

A new question: when should effects take place?

Example: after staging `f` ...

```
let ncalls = ref 0
let f x y = incr ncalls; g x y
```

...should the call counter be bumped during code generation ...

```
let f x = .< fun y → .~(incr ncalls; g x .<y>.) >.
```

...or during code execution?

```
let f x = .< fun y → incr ncalls; .~(g x .<y>.) >.
```

Effects are **harmful** (w.r.t. staging)

What does the following code do?

```
exception Var of int code

try
   .< fun x → .~( raise (E (.< x >.)) ) >.
with Var v →
   .< fun y → .~(v) >.
```

Effects are helpful (w.r.t. staging)

```
val genlet : 'a code → 'a code

val let_locus : (unit → 'a code) → 'a code
```

```
let_locus (fun () →
 .<
    a + b + .~(genlet .<c + d>.)
   >.)
```

⇝

```
.<
 let x = c + d in
 a + b + x
 >.
```

let insertion: a simple implementation

```
effect GenLet : 'a code → 'a code

let genlet v = perform (GenLet v)

let let_locus body =
  try body ()
  with effect (GenLet v) k →
    .< let x = .~v in .~(continue k .< x >.)>.
```

let insertion at the outermost valid point

```
let is_well_scoped c =
  try ignore .< (.~c; ()) >.; true
  with _ → false

let genlet v =
  try perform (GenLet v)
  with Unhandled → v

let let_locus body =
  try body ()
  with effect (GenLet v) k when is_well_scoped v →
    match perform (GenLet v) with
    | v → continue k v
    | exception Unhandled → .< let x = .~v in .~(
        continue k .< x >.)>.
```

let insertion for sharing

Example:

```
let x = .< y * y >. in .< .~x * .~x >.
```

becomes

```
.< (y * y) * (y * y) >.
```

but

```
let x = genlet .< y * y >. in .< .~x * .~x >.
```

becomes

```
.< let v = y * y in ... v * v >.
```

Improving `pow` (for the last time)

Problem: the generated code is still inefficient.

```
.< x * x * x * x >.
```

Aim: reduce the multiplications by repeated squaring

```
.< let y = x * x in y * y >.
```

Constraint: leave the `pow` code unchanged

```
let rec pow : {N:MON} → N.t → int → N.t  =
  fun {N:MON} x n →
    if n = 0 then N.one
    else x * pow x (n - 1)
```

Plan: improved data representation, `let` insertion

pow: our best representation so far

Approach: leave code construction *as late as possible*

(Remember: `code` cannot be optimized!)

Partially-static integers (our best representation so far!)

```
(* s * d *)
type t = { sta: int; dyn: int code }
```

Multiplication for partially-static integers:

```
let mul x y = match x.sta * y.sta, x.dyn, y.dyn with
    0, _, _ → { sta = 0; dyn = None }
  | s, None, d
  | s, d, None → { sta = s; dyn = d }
  | s, Some d1, Some d2 →
    {sta = s; dyn = Some (.< .~d1 * .~d2 >.)}
```

Problems with the partially-static implementation

Static components are multiplied statically

```
{sta=3; dyn=None} <*> {sta=4; dyn=Some .<x>.}
⇝
{sta=12; dyn=Some .<x>.}
```

Dynamic components are multiplied dynamically

```
{sta=3; dyn=Some .<x>.} <*> {sta=4; dyn=Some .<x>.}
⇝
{sta=12; dyn=Some .<x*x>.}
```

Dynamic components are never inspected

```
{sta=12; dyn=Some .<x*x>.} <*> {sta=12; dyn=Some .<x*x>.}
⇝
{sta=144; dyn=Some .<x*x*x*x>.}
```

pow: delaying code construction

Idea

Delay code construction to the last moment.

Our final representation (for this week)

```
type var = Var of int code * int
type t = { sta: int; dyn: (var * int) list }
       (* s × d1^s1 × d2^s2 ... × dn^sn *)
```

Multiplication

$$
\begin{array}{ll}
 & \mathtt{s} \times \mathtt{d}_1^{s_1} \times \mathtt{d}_2^{s_2} \ldots \times \mathtt{d}_n^{s_n} \\
\texttt{<*>} & \mathtt{t} \times \mathtt{d}_1^{t_1} \times \mathtt{d}_2^{t_2} \ldots \times \mathtt{d}_n^{t_n} \\
\rightsquigarrow & (\mathtt{s}\times\mathtt{t}) \times \mathtt{d}_1^{s_1+t_1} \times \mathtt{d}_2^{s_2+t_2} \ldots \times \mathtt{d}_n^{s_n+t_n}
\end{array}
$$

pow: delaying code construction

Our final representation (for this week)

```
type var = Var of int code * int
type t = { sta: int; dyn: (var * int) list }
```

$$\texttt{(* s} \times \texttt{d}_1^{s_1} \times \texttt{d}_2^{s_2} \ldots \times \texttt{d}_n^{s_n} \texttt{ *)}$$

Code generation

$$\texttt{cd (s} \times \texttt{d}_1^{s_1} \times \texttt{d}_2^{s_2} \ldots \times \texttt{d}_n^{s_n}\texttt{)}$$

$\rightsquigarrow$

$$\texttt{.< let x}_1 \texttt{ = d}_1 \times \texttt{d}_1 \texttt{ in}$$
$$\texttt{.\~{}(cd (s} \times \texttt{x}_1^{s_1/2} \times \texttt{d}_2^{s_2} \ldots \times \texttt{d}_n^{s_n}\texttt{)) >.}$$

$\rightsquigarrow$

$$\texttt{.< let x}_1 \texttt{ = d}_1 \times \texttt{d}_1 \texttt{ in}$$
$$\texttt{let x}_2 \texttt{ = x}_1 \times \texttt{x}_1 \texttt{ in}$$
$$\texttt{.\~{}(cd (s} \times \texttt{x}_2^{s_1/4} \times \texttt{d}_2^{s_2} \ldots \times \texttt{d}_n^{s_n}\texttt{)) >.}$$

$\rightsquigarrow$

...

pow: improved code

```
# .< fun x →  .~(let_locus @@ fun () →
                    cd (pow (var .<x>.) 8)) >.
  - : (int → int) code =
 .< fun x →
    let x1 = x * x in
    let x2 = x1 * x1 in
    let x3 = x2 * x2 in
      x3 >.
```

Recursion revisited

Handling recursion so far: unrolling

```
let rec pow : {N:MON} → N.t → int → N.t  =
  fun {N:MON} x n →
    if n = 0 then N.one
    else x * pow x (n - 1)
```

```
# .< fun x →  .~(dyn (pow (cd .<x>.) 5)) >.
- : (int → int) code = .< fun x → x * x * x * x * x >.
```

But unrolling is not always appropriate…

Unrolling with a dynamic inductive parameter

Example: `pow` with a *static base* and *dynamic exponent*:

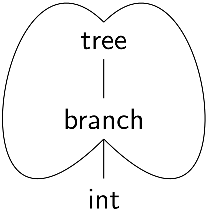
```
let rec pow : int → int code → int code =
  fun x n →
   .< if  .~n = 0 then 1
        else x *  .~(pow x .< .~n - 1 >.) >.
```

What is the result of the following call?

```
pow 2 .<3>.
```

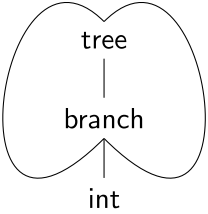
Unrolling with cyclic / infinite structure

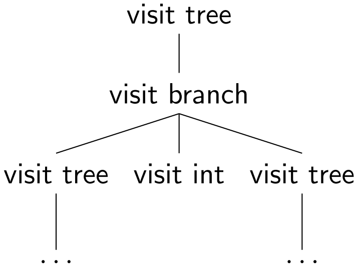
```
type tree =
  Empty
| Branch of tree * int * tree
```



Unrolling with cyclic / infinite structure

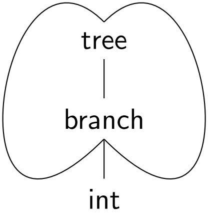
```
type tree =
  Empty
| Branch of tree * int * tree
```

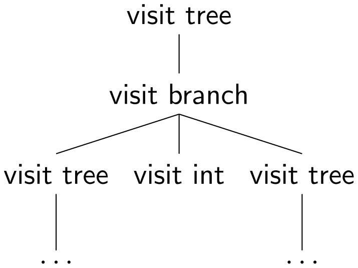




Unrolling with cyclic / infinite structure

```
type tree =
  Empty
| Branch of tree * int * tree
```





(and sometimes unrolling just generates enormous code)

Alternatives to unrolling: memoization

```
let rec fib = function
    0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```

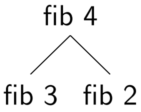
Alternatives to unrolling: memoization

```
let rec fib = function
    0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```

fib 4

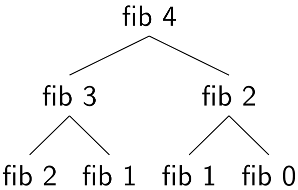
Alternatives to unrolling: memoization

```
let rec fib = function
    0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```



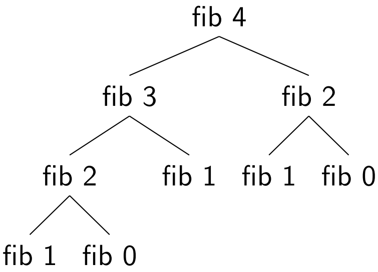
Alternatives to unrolling: memoization

```
let rec fib = function
    0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```



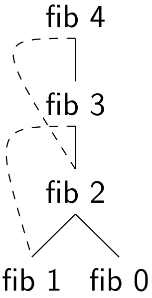
Alternatives to unrolling: memoization

```
let rec fib = function
    0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```



Alternatives to unrolling: memoization

```
let table = ref []

let rec fib n =
 try List.assoc n !table
 with Not_found →
  let r = fib_aux n in
  table := (n, r) :: !table;
  r
and fib_aux = function
    0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```



Memoization, factored

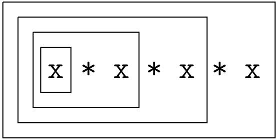
```
val memoize : (('a → 'b) → ('a → 'b)) → 'a → 'b

let memoize f n =
  let table = ref [] in
  let rec f' n =
    try List.assoc n !table
    with Not_found →
      let r = f f' n in
      table := (n, r) :: !table;
      r
  in f' n


let open_fib fib = function
  0 → 0
| 1 → 1
| n → fib (n - 1) + fib (n - 2)

let fib = memoize open_fib
```

A difficulty:

`let rec` insertion

`let rec` insertion: the problem

Expressions are built from smaller expressions:




```
.< .~(.< .~(.< .~(.< x >.) * x >.) * x >.) * x >.
```

Binding groups are *not* built from smaller binding groups:

```
let rec x_1 = e_1
    and x_2 = e_2
    ...
    and x_n = e_n
```

let rec insertion: Landin's knot

```
let rec f₁ = fun x → e₁
    and f₂ = fun x → e₂
    ...
    and fₙ = fun x → eₙ
     in e
```

becomes

```
let f₁ = ref dummy in
let f₂ = ref dummy in
...
let fₙ = ref dummy in

   f₁ := fun x → e₁[fᵢ:=!fᵢ];
   f₂ := fun x → e₂[fᵢ:=!fᵢ];
   ...
   fₙ := fun x → eₙ[fᵢ:=!fᵢ];
   e![fᵢ:=!fᵢ]!
```

Let bindings and sequencing *are* built from smaller expressions:

```
(let f₁ = ref dummy in
 (let f₂ = ref dummy in
  ...
  (let fₙ = ref dummy in
   ...) ...))

(f₁ := e₁;
 (f₂ := e₂;
  ...
  (fₙ := eₙ) ...))
```

let rec insertion: the solution

```
val genletrec : (('a → 'b) code → ('a → 'b) code) →
                 ('a → 'b) code

let genletrec k =
 let r = genlet (.< ref dummy >.) in
   genlet (.<.~r := .~(k .< ! .~r >.) >.);
  .< ! .~r >.
```

Staging
generic programming

What is generic programming?

```
type 'a tree =
    Empty : 'a tree
  | Branch : 'a tree * 'a * 'a tree → 'a tree

module type SHOW = sig
 type t
 val show : t → string
end

implicit module Show_tree {A:SHOW} = struct
  let rec show = function
    | Empty → "Empty"
    (* ... *)
end
```

Aim: eliminate "boilerplate" like `Show_tree`
(code that simply follows type structure)

Scrap Your Boilerplate: 3 ingredients

1. The `TYPEABLE` interface: **generic type equality**
2. The `DATA` interface: **shallow traversals** of data structures
3. **Recursive schemes**, such as `everywhere`, `gshow`, ...

SYB ingredient 1: type equality tests

```
val (=~=) : {A: TYPEABLE} → {B: TYPEABLE} →
  (A.t, B.t) eql option
```

```
val cast : {A: TYPEABLE} → {B: TYPEABLE} →
   (A.t → B.t) option
```

Implementation: extensible GADTs:

```
type _ type_rep = ..

type _ type_rep +=
  Int : int type_rep

let eqty_int :
  type b. b type_rep → (int, b) eql option =
  function Int → Some Refl | _ → None
```

SYB ingredient 2: shallow traversals

```
module type rec DATA =
sig
 type t
 val gmapQ : (∀D.{D: DATA} →D.t → 'u) → t → 'u list
 (* ... *)
end

implicit module rec DATA_tree {A: DATA}
  : DATA with type t = A.t tree =
struct
  type t = A.t tree

  let gmapQ q = function
      Empty → []
    | Branch (l, v, r) → [q l; q v; q r]
  (* ... *)
end
```

SYB ingredient 3: generic schemes

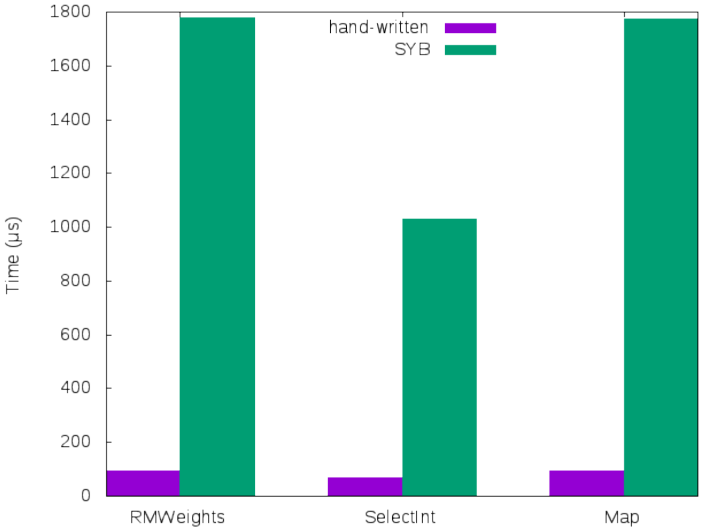
```
val everywhere : (∀D.{D: DATA} → D.t → D.t) →
                  {T: DATA} → T.t → T.t

let rec everywhere f {D:DATA} x =
  f (gmapT (everywhere f) x)
```

```
val gshow : {T: DATA} → T.t → string

let rec gshow {D:DATA} v =
  show_constructor (constructor v, gmapQ gshow v)
```

SYB in action

```
# everywhere (mkT succ) [(1, true); (3, false)]
- : (int * bool) list = [(2, true); (4, false)]

# gshow (Branch (Empty, 3, Empty))
- : string = "(Branch (Empty, 3, Empty))"
```

SYB is *slow*



hand-written
SYB
Time (µs)
0
200
400
600
800
1000
1200
1400
1600
1800
RMWeights
SelectInt
Map

Why is SYB slow?

- Type equality tests are slow because
  types are (mostly) static, but checks are dynamic.
- Shallow traversals are slow because of
  polymorphic calls and run-time dictionaries.
- Generic traversals are slow because
  open (polymorphic!) recursion involves indirect calls.

Improving SYB's performance

with

staging

Binding-time analysis

```
val gshow : {T: DATA} → T.t → string
```

Type representations are **static**. **Values** are **dynamic**.

SYB uses type representations to traverse values.

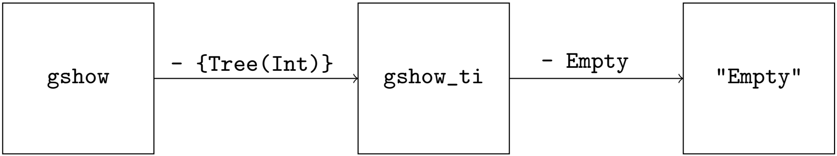
We'll use type representations to generate code.

Goal: generate code that contains no `TYPEABLE` or `DATA` values.

Staging SYB: overview

Plan: turn generic functions like `gshow` into *code generators*.

Call `gshow {Tree{Int}}` to generate a specialized printing function.



Three-pronged approach

1. The TYPEABLE interface
Move checks to code-generation time; otherwise unchanged

2. The DATA interface
Stage, treating type representation dictionaries as static

3. Recursive schemes: `everywhere`, `gshow`, ...
Close and monomorphize the recursion

Staging gmapQ

```
module type rec DATA =
sig
 type t
 val gmapQ : (∀D.{D: DATA} →D.t code → 'u code) →
                   t code → 'u list code
 (* ... *)
end

implicit module DATA_tree {A:DATA}
  : DATA with type t = A.t tree =
struct
  (* ... *)

  let gmapQ q l =
  .< match .~l with
       Empty → []
     | Branch (l, v, r) →
         [.~(q .<l>.); .~(q .<v>.); .~(q .<r>.)]
   >.
end
```

Memoization: Typed maps

```
type _ t =
  Nil : 'a t
| Cons : {T.TYPEABLE} * (T.t → 'a) code * 'a t → 'a t

val new_map : unit → 'a t ref

val add :
  {T:TYPEABLE} → (T.t → 'a) code → 'a t ref → unit

val lookup :
  {T:TYPEABLE} → 'a t → (T.t → 'a) code option
```

Memofix combinators

```
val gfixQ : ((∀A.{A:DATA} →A.t code → 'u code) →
              (∀B.{B:DATA} →B.t code → 'u code)) →
               {C:DATA} → C.t code → 'u code

let gfixQ f =
 let tbl = new_map () in
 let rec result {D: DATA} x =
   match lookup {D.Typeable} !tbl with
     Some g → .< .~g .~x >.
   | None →
      let g = genletrec
         (fun self →
            push tbl self;
          .< fun y → .~(f result .<y>.) >.)
       in .< .~g .~x >.
  in result
```

Generation and instantiation

```
val generateQ : {D:DATA} →
                ({T:DATA} → T.t code → 'u code) →
                (D.t → 'u) code

let generateQ {D:DATA} q =
  let_locus (fun () → .< fun x → .~(q .<x>.) >.)



val instantiateQ : {D:DATA} →
                   ({T:DATA} → T.t code → 'u code)
                       →
                   (D.t → 'u)

let instantiateQ {D: DATA} q =
  Runcode.run (generateQ q)
```

Generated code for gshow

```
let show_tree = ref dummy in
let show_branch = ref dummy in
let show_int = ref dummy in
let _ = show_int :=
 fun i →
  "("^ string_of_int i ^ String.concat " " [] ^")" in
let _ = show_branch :=
  fun b →
   "("^ "(,)"^
        ((String.concat " "
          (let (l,v,r) = b in
           [!show_tree l; !show_int v; !show_tree r]))
        ^")") in
let _ = show_tree :=
  (fun t →
     "("^ ((match t with Empty → "Empty"
                       | Branch _ → "Branch") ^
         ((String.concat " "
              (match t with
               | Empty → []
               | Branch b → [!show_branch b])) ^")"))) in
!show_tree
```

Staged SYB performance