Multi-stage programming

Part I: Static and Dynamic

Jeremy Yallop

University of Cambridge

International Summer School on Metaprogramming

August 2016

Themes and aims

Abstraction vs performance

(Guilt, regret and shame)

Automatic vs manual

(Partial evaluation vs staging)

Old ideas in new settings

(Optimization passes as libraries)

Following along

```
$ opam switch 4.02.1+modular-implicits-ber
 [...]
$ eval `opam config env`
```

or

Online: `http://yallop.github.io/iocamljs/summer.html`
(linked from lecture notes page)

Abstraction wants to be free

```
let pow2 x = x * x                 (* x^2 *)
let pow3 x = x * x * x             (* x^3 *)
let pow5 x = x * x * x * x * x     (* x^5 *)
```

```
let rec pow x n =                  (* x^n *)
  if n = 0 then 1
  else x * pow x (n - 1)
```

Goal: high-level abstraction, low-level performance

We'd like to write this:

```
let rec pow x n =                    (* x^n *)
  if n = 0 then 1
  else x * pow x (n - 1)
```

but have it perform like this:

```
let pow2 x = x * x                   (* x^2 *)
let pow3 x = x * x * x               (* x^3 *)
let pow5 x = x * x * x * x * x       (* x^5 *)
```

Goal: high-level abstraction, low-level performance

We'd like to write this:

```
let rec pow x n =                    (* x^n *)
  if n = 0 then 1
  else x * pow x (n - 1)
```

but have it perform ~~like~~ better than this:

```
let pow2 x = x * x                   (* x^2 *)
let pow3 x = x * x * x               (* x^3 *)
let pow5 x = x * x * x * x * x       (* x^5 *)
```

Starting point: *raise* the level of abstraction

```
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)
```

Our generalized `pow` can be used for (e.g.) strings:

`pow "a" 5` ⇝ `"aaaaa"`

Modular implicits

```
module type MON =
sig
  type t
  val one : t
  val mul : t → t → t
end
```

Interface

```
implicit module MON_int =
struct
  type t = int
  let one = 1
  let mul x y = x * y
end
```

Instance

```
let ( <*> ) {N: MON} x y = N.mul x y
```

Overloaded function

```
2 <*> N.one
```

Call

Calling `pow`

```
# pow 2 5;;
- : int = 32
```

MetaOCaml: quotes and splices

MetaOCaml: multi-stage programming with code quoting.

Stages: current (available now) and delayed (available later).
(Also double-delayed, triple-delayed, etc.)

Brackets

```
.< e >.
```

Escaping (within brackets)

```
.~ e
```

Running code

```
!. e
```

Cross-stage persistence

```
.< x >.
```

MetaOCaml typing rules

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

$$\frac{\Gamma \vdash^{n+} \texttt{e} : \tau}{\Gamma \vdash^{n} .\!<\texttt{e}>\!. : \tau\ \text{code}}\ \text{T-bracket} \qquad \frac{\Gamma^{+} \vdash^{n} \texttt{e} : \tau\ \text{code}}{\Gamma \vdash^{n} \texttt{!. e} : \tau}\ \text{T-run}$$

$$\frac{\Gamma \vdash^{n} \texttt{e} : \tau\ \text{code}}{\Gamma \vdash^{n+} .\tilde{}\ \texttt{e} : \tau}\ \text{T-escape} \qquad \frac{\Gamma(x) = \tau^{(n-m)}}{\Gamma \vdash^{n} \texttt{x} : \tau}\ \text{T-var}$$

MetaOCaml quoting: basic examples

```
.< 3 >.

.< 1 + 2 >.

.< [1; 2; 3] >.

let x = 3 in .< x + y >.

.< fun x -> x >.

.< (.~f)3 >.

.< .~(f 3) >.

.< fun x -> .~(f .< x >.) >.
```

Learning
from
mistakes

Learning from mistakes: I

```
.< 1 + "two" >.
```

Learning from mistakes: I

```
# .< 1 + "two" >.;;
Characters 7-12:
  .< 1 + "two" >.;;
          ^^^^^
```

```
Error: This expression has type string but an
    expression was expected of type int
```

Learning from mistakes: II

```
.< fun x → .~( x ) >.
```

Learning from mistakes: III

```
# .< fun x → .~( x ) >.;;
Characters 14-19:
  .< fun x → .~( x ) >.;;
                ^^^^^
```

```
Wrong level: variable bound at level 1 and
    used at level 0
```

Learning from mistakes: IV

```
let x = .< 3 >. in .~x
```

Learning from mistakes: IV

```
# let x = .< 3 >. in .~x;;
Characters 22-23:
  let x = .< 3 >. in .~x;;
                     ^

Wrong level: escape at level 0
```

Error: running open code

```
.< fun x → .~(!. .< x >. ) >.
```

Error: running open code

```
# .< fun x → .~(!. .<x>. ) >.;;
Exception:
Failure
 "The code built at Characters 7-8:\n
 .< fun x → .~(!. .<x>. ) >.;;\n
       ^\n
 is not closed: identifier x_2 bound at
 Characters 7-8:\n
 .< fun x → .~(!. .<x>. ) >.;;\n
       ^\n
 is free".
```

Staging `pow`

Staging pow

```
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)


implicit module MON_intcode = struct
  type t = int code
  let one = .<1>.
  let mul x y = .< .~x * .~y >.
end


# let pow5 = .< fun x → .~(pow .<x>. 5) >.;;



# let pow5' = Runcode.run pow5;;


# pow5' 2;;
```

Staging pow

```
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)


implicit module MON_intcode = struct
  type t = int code
  let one = .<1>.
  let mul x y = .< .~x * .~y >.
end


# let pow5 = .< fun x → .~(pow .<x>. 5) >.;;
- : (int → int) code =
  .< fun x → x * (x * (x * (x * (x * 1))))>.

# let pow5' = Runcode.run pow5;;
val pow5' : int → int = <fun>

# pow5' 2;;
- : int = 32
```

Binding-time analysis

Classify **variables**: dynamic / static

```
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)
```

static: `N`, `n`
dynamic: `x`

Classify **expressions**: static (no dynamic dependencies) / dynamic

```
    if n = 0 then N.one
    else x <*> pow x (n - 1)
```

static: `n = 0`, `n - 1`, `N.one`, `pow x (n - 1)`
dynamic: `x <*> pow x (n - 1)`

Goal: reduce static expressions during code generation.

The idealized staging process

1. Write the program as usual:

   ```
   val program : t_STA → t_DYN → t
   ```

The idealized staging process

1. Write the program as usual:

```
val program : t_STA → t_DYN → t
```

2. Add staging annotations:

```
val staged_program : t_STA → t_DYN code → t code
```

The idealized staging process

1. Write the program as usual:

```
val program : t_STA → t_DYN → t
```

2. Add staging annotations:

```
val staged_program : t_STA → t_DYN code → t code
```

3. Compile using `back`:

```
val back: ('a code → 'b code) → ('a → 'b) code
val code_generator : t_STA → (t_DYN → t) code
```

The idealized staging process

1. Write the program as usual:

```
val program : t_STA → t_DYN → t
```

2. Add staging annotations:

```
val staged_program : t_STA → t_DYN code → t code
```

3. Compile using `back`:

```
val back: ('a code → 'b code) → ('a → 'b) code
val code_generator : t_STA → (t_DYN → t) code
```

4. Construct static inputs:

```
val s : t_STA
```

The idealized staging process

1. Write the program as usual:

   ```
   val program : t_STA → t_DYN → t
   ```

2. Add staging annotations:

   ```
   val staged_program : t_STA → t_DYN code → t code
   ```

3. Compile using `back`:

   ```
   val back: ('a code → 'b code) → ('a → 'b) code
   val code_generator : t_STA → (t_DYN → t) code
   ```

4. Construct static inputs:

   ```
   val s : t_STA
   ```

5. Apply code generator to static inputs:

   ```
   val specialized_code : (t_DYN → t) code
   ```

The idealized staging process

1. Write the program as usual:

```
val program : t_STA → t_DYN → t
```

2. Add staging annotations:

```
val staged_program : t_STA → t_DYN code → t code
```

3. Compile using `back`:

```
val back: ('a code → 'b code) → ('a → 'b) code
val code_generator : t_STA → (t_DYN → t) code
```

4. Construct static inputs:

```
val s : t_STA
```

5. Apply code generator to static inputs:

```
val specialized_code : (t_DYN → t) code
```

6. Run specialized code to build a specialized function:

```
val specialized_function : t_DYN → t
```

Improving binding times

Improving binding times: dynamic infection

```
char_of_int (if bit = 0 then 0 else 0xFF)
```

Improving binding times: dynamic infection

```
char_of_int (if bit = 0 then 0 else 0xFF)
```

Static: `char_of_int`, `0`, `0xFF`
Dynamic: `bit`, `bit = 0`, `if bit = 0 then 0 else 0xFF`

CPS conversion improves binding times

```
let k2 v = k (char_of_int v) in
if bit = 0 then k2 0 else k2 0xFF
```

Static: `char_of_int`, `v`, `k2`, `k2 0`, `k2 0xFF`

Dynamic: `bit`, `bit = 0`, `if bit = 0 then k 0 else k 0xFF`

The Trick

```
char_of_int (if bit = 0 then 0 else 0xFF)
```

Insight

`bit = 0` is **dynamic** but it has **only two possible values**

The Trick

```
char_of_int (if bit = 0 then 0 else 0xFF)
```

```
match bit = 0 with
  false → char_of_int (if false then 0 else 0xFF)
| true → char_of_int (if true then 0 else 0xFF)
```

The Trick

```
char_of_int (if bit = 0 then 0 else 0xFF)
```

```
match bit = 0 with
  false → char_of_int (if false then 0 else 0xFF)
| true → char_of_int (if true then 0 else 0xFF)
```

```
.< match .~(bit) = 0 with
  false → .~(char_of_int (if false then 0 else 0xFF))
| true → .~(char_of_int (if true then 0 else 0xFF)) >.
```

η expansion (does The Trick)

```
f ≡ fun x → f x
```

functions

```
e ≡ (fst e, snd e)
```

products

```
e[x:=y] ≡ if y
          then e[x:=true]
          else e[x:=false]
```

booleans

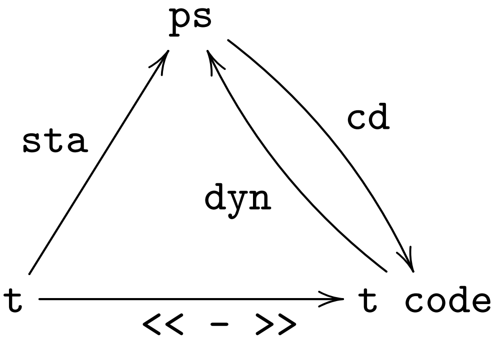
```
e[x:=v] ≡ match v with
          | L x → e[x:=L x]
          | R x → e[x:=R x]
```

sums

Improving binding times

via data representation

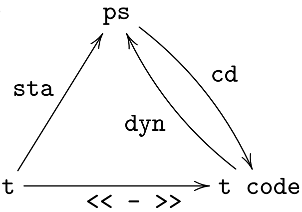
Possibly-static data (+ online partial evaluation)



Generic operations on possibly-static data

```
type 'a sd =
  Sta : 'a → 'a sd
| Dyn : 'a code → 'a sd
```

```
let sta s = Sta s
```




```
let dyn d = Dyn d

let cd = function
 | Sta s → .< s >.
 | Dyn d → d
```

Arithmetic with possibly-static data

```
implicit module Num_ps_int =
struct
  type t = int sd

  let one = sta 1

  let mul x y = match x, y with
   | Sta l   , Sta r → Sta (l * r)
   | Sta 1, x
   | x      , Sta 1 → x
   | Sta 0, _
   | _      , Sta 0 → Sta 0
   | x, y → Dyn .< .~(cd x) * .~(cd y) >.
end
```

Question: what effect will this have on `pow`?

Calling `pow` with possibly-static data

```
.< fun x → .~(cd (pow (dyn .<x>.) 5)) >.
```

```
.< fun x → x * x * x * x * x >.
```

A small problem: associativity

```
(sta 3 <*> sta 4) <*> dyn .< 6 >.

sta 3 <*> (sta 4 <*> dyn .< 6 >.)
```

Questions

Which expressions are static?

What code will be generated?

A small problem: associativity

```
(sta 3 <*> sta 4) <*> dyn .< 6 >.

sta 3 <*> (sta 4 <*> dyn .< 6 >.)
```

Questions

Which expressions are static?

What code will be generated?

What's gone wrong?

Partially-static data

Progressively refining numeric representations:

(Definitely) static

```
type t = int
```

(Definitely) dynamic

```
type t = int code
```

Possibly-static (static *or* dynamic)

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

Partially-static (static × dynamic)

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

Partially-static data: generic operations

```
module type PS =
sig
 type t
 type ps

 val sta : t → ps

 val dyn : t code → ps

 val cd : ps → t code




end
```

```
implicit module PS_sd_int =
struct
 type t = int
 type ps = { sta: int;
             dyn: int code option }
 let sta s =
   { sta = s; dyn = None }
 let dyn d =
   { sta = 1; dyn = Some d}
 let cd = function
   | {sta=1; dyn=Some d} → d
   | {sta=s; dyn=Some d} →
     .< s * .~d >.
   | {sta=s} → .<s >.
end
```

Arithmetic with partially-static data

```
implicit module MON_int_ps =
struct
 type t = { sta: int; dyn: int code option }

 let one = { sta = 1; dyn = Some .< 1 >. }

 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 >.)}
end
```

Associativity fixed

```
# (sta 4 <*> sta 5) <*> dyn .< 6 >.;;
- : t = {sta = 20; dyn = Some .< 6 >. }

# sta 4 <*> (sta 5 <*> dyn .< 6 >.);;
- : t = {sta = 20; dyn = Some .< 6 >. }
```

A further problem: insufficient sharing

```
# let pow8 = .< fun x → .~(cd (pow (dyn .< x >.) 8)) >.)
val pow8 : (int → int) code =
.< fun x → x * (x * (x * (x * (x * (x * (x * x))))))) >.
```

Q: How can we reduce the number of multiplications?

(To be continued. . .)

Partially-static compound data

Complex numbers

```
module type COMPLEX =
sig
  type t
  type elem
  val re : t → elem
  val im : t → elem
  val mk : elem → elem → t
  (* ... *)
end
```

Partially-static complex numbers

```
implicit module COMPLEX_float = struct
  type elem = float
  type t = { re: elem; im: elem }
  let re {re} = re
  let im {im} = im
  let mk re im = { re; im }
  (* ... *)
end
```

Complex numbers

```
implicit module COMPLEX_ps_float = struct
  type elem = float sd
  type t = { re: elem; im: elem }
  let re {re} = re
  let im {im} = im
  let mk re im = { re; im }
  (* ... *)
end
```

Partially-static complex numbers

Operations on partially-static complex numbers

```
module type COMPLEX =
sig
  type t
  type elem
  val re : t → elem
  val im : t → elem
  val mk : elem → elem → t
  (* ... *)
end
```

```
mk ((re x <*> re y) <-> (im x <*> im y))
   ((re x <*> im y) <+> (im x <*> re y))
```

Partially-static complex numbersm, improved

```
implicit module COMPLEX_ps_float' = struct
  type elem = float sd
  type t = Sta of { re: elem; im: elem }
         | Dyn of COMPLEX_float code
  let re = function
      Sta {re} → re
    | Dyn c → dyn .< re .~c >.
  (* ... *)
end
```

Partially-static complex numbers, improved

```
mk ((re x <*> re y) <-> (im x <*> im y))
   ((re x <*> im y) <+> (im x <*> re y))
```

Problem: loss of sharing (again!)

Partially-static complex numbersm, improved

```
implicit module COMPLEX_ps_float' = struct
  type elem = float sd
  type t = Sta of (elem * elem)
         | Dyn of COMPLEX_float.t code
  let re = function
      Sta (re,im) → re
    | Dyn c → dyn .< re .~c >.
  (* ... *)
end
```

Partially-static complex numbers, improved

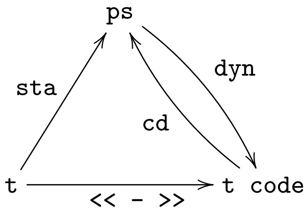
```
mk ((re x <*> re y) <-> (im x <*> im y))
   ((re x <*> im y) <+> (im x <*> re y))
```

Problem: loss of sharing (again!)
. . . but we know the structure, even of dynamic values!

Static inspection of dynamic data (η revisited)

```
let split : t → float sd * float sd = function
    Sta (re, im) → (re, im)
  | Dyn d → (Dyn .< re .~(v) >., Dyn .< im .~(v) >.)
```

```
let xre, xim = split x and yre, yim in split y in
 mk ((xre <*> yre) <-> (xim <*> yim))
    ((xre <*> yim) <+> (xim <*> yre))
```


ps
sta
dyn
cd
t
t code
<< - >>