

## Last time: effects

`effect E : t`

This time: staging

.< e >.

# Review: abstraction

## Lambda abstraction

$\lambda x : A.M$

$\Lambda A :: K.M$

$\lambda A :: K.B$

## First-class $\forall$ and $\exists$

```
type t = { f: 'a. ... }
```

```
type t = E : 'a s → t
```

## Modular abstraction

```
module F(X : T) = ...
```

```
let f {X:T} = ...
```

## Abstraction of type equalities

$a \equiv b$

## Interfaces to computation

```
m  $\gg$ = k    f  $\otimes$  p  
          effect E:t
```

## Abstraction over data shape

```
val show :  
  {D:DATA} → 'a → string
```

# The cost of ignorance

## Fewer opportunities for optimization

```
let both_eq1 : int * int → int * int → bool =  
  fun (x1, y1) (x2, y2) →  
    x1 = x2 && y1 = y2
```

```
let both_eq2 : (int → int → bool) →  
  int * int → int * int → bool =  
  fun eq (x1, y1) (x2, y2) →  
    eq x1 x2 && eq y1 y2
```

```
both_eq2 (fun x y → x = y)
```

```
type eq = { eq: 'a. 'a → 'a → bool }  
let both_eq {eq} (x1, y1) (x2, y2) =  
  eq x1 x2 && eq y1 y2
```

# The cost of ignorance

## Interpretative overhead

```
let print_int_pair (x,y) =  
  print_char '(';  
  print_int x;  
  print_char ',';  
  print_int y;  
  print_char ')'
```

```
let print_int_pair2 (x,y) =  
  Printf.sprintf "(%d,%d)" x y
```

```
let print_int_pair3 (x,y) =  
  print_string (gshow (pair int int) (x, y))
```

## Abstraction wants to be free

```
let pow2 x = x * x           (* x2 *)  
let pow3 x = x * x * x      (* x3 *)  
let pow5 x = x * x * x * x * x (* x5 *)
```

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

```
val pow : int → int → int
```

## Power, staged

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

## Power, staged

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



## Power, staged

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

## Power, staged

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

## Power, staged

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

```
val pow : int code → int → int code
```

```
let pow_code n = .< fun x → .~(pow .<x>. n) >.
```

```
val pow_code : int → (int → int) code
```

```
# pow_code 3;;
```

```
.<fun x → x * x * x * 1>.
```

## Power, staged

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

val pow : int code → int → int code

let pow_code n = .< fun x → .~(pow .<x>. n) >.

val pow_code : int → (int → int) code

# pow_code 3;;
.<fun x → x * x * x * 1>.

# let pow3' = !. (pow_code 3);;
val pow3' : int → int = <fun>
```

## Power, staged

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

val pow : int code → int → int code

let pow_code n = .< fun x → .~(pow .<x>. n) >.

val pow_code : int → (int → int) code

# pow_code 3;;
.<fun x → x * x * x * 1>.

# let pow3' = !. (pow_code 3);;
val pow3' : int → int = <fun>

# pow3' 4;;
- : int = 64
```

# MetaOCaml basics

## Quoting

```
let x = "w" in
let y = "x" in
  print_string (x ^ y)
```

```
let x = "w" in
let y = x in
  print_string ("x ^ y")
```

```
let x = "w" in
let y = x in
  print_string (x ^ y)
```

```
let x = "w" in
let y = x in
  print_string ("x" ^ y)
```

**Quoting prevents evaluation.**

## Quoting code

**MetaOCaml:** multi-stage programming with code quoting.

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

**Brackets**

`.<e>.`

**Running code**

`!. e`

**Escaping** (within brackets)

`.~e`

**Cross-stage persistence**

`.<x>.`

**Goal:** generate a specialized program with better performance



## Quoting and escaping: some examples

`.<3 >.`

`.<1 + 2 >.`

`.<[1; 2; 3] >.`

`.<x + y >.`

`.<fun x → x >.`

`.<(.~f) 3 >.`

`.<~(f 3)>.`

`.<fun x → ~(f .<x >.) >.`

## Quoting: typing

$$\Gamma \vdash^n e : \tau$$

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^n \langle e \rangle : \tau \text{ code}} \text{ T-bracket}$$

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

$$\frac{\Gamma \vdash^n e : \tau \text{ code}}{\Gamma \vdash^{n+1} \sim e : \tau} \text{ T-escape}$$

$$\frac{\Gamma(x) = \tau^{(n-m)}}{\Gamma \vdash^n : \tau} \text{ T-var}$$

## Quoting: open code

### Open code (supports symbolic computation)

```
let pow_code n = .< fun x → .~(pow .<x>. n) >.
```

### Cross-stage persistence

```
let print_int_pair (x,y) =  
  Printf.printf "(%d,%d)" x y
```

```
let pairs = .< [(3, 4); (5, 6)] >.
```

```
.< List.iter print_int_pair .~pairs >.
```

## Quoting: scoping

Scoping is **lexical**, just as in OCaml.

```
.< fun x → .~( let x = 3 in .<x>. ) >.
```

```
let x = 3 in .< fun x → .~( .<x>. ) >.
```

MetaOCaml renames variables to avoid clashes:

```
.< let x = 3 in  
  .~(let y = .<x>. in  
    .< fun x → .~y + x >.) >.
```

## Quoting: scoping

Scoping is **lexical**, just as in OCaml.

```
.< fun x → .~( let x = 3 in .<x>. ) >.
```

```
let x = 3 in .< fun x → .~( .<x>. ) >.
```

MetaOCaml renames variables to avoid clashes:

```
# .< let x = 3 in
    .~(let y = .<x>. in
        .< fun x → .~y + x >.) >;;
- : (int → int) code =
.<let x_1 = 3 in fun x_2 → x_1 + x_2>.
```

# Learning from mistakes

## Error: quoting nonsense

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

## Error: quoting nonsense

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

```
Characters 7-12:
```

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

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



## Error: looking into the future

```
.< fun x → .~( x ) >.
```

## Error: looking into the future

```
# .<fun x → .~( x ) >.;;
```

```
Characters 14-19:
```

```
  .<fun x → .~( x ) >.;;  
                ^^^^^
```

Error: A variable that was bound within brackets is  
used outside brackets

for example: .<fun x -> .~(foo x)>.

Hint: enclose the variable in brackets,

as in: .<fun x -> .~(foo .<x>.)>.;;

## Error: escape from nowhere

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

## Error: escape from nowhere

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

```
Characters 22-23:
```

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

```
Error: An escape may appear only within brackets
```

## 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".
```

# Learning by doing

## The staging process, idealized

1. Write the program as usual:

```
val program : t_sta → t_dyn → t
```



## The staging process, idealized

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 staging process, idealized

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)
```

## The staging process, idealized

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)
```

4. Construct static inputs:

```
val s : t_sta
```

## The staging process, idealized

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)
```

4. Construct static inputs:

```
val s : t_sta
```

5. Apply code generator to static inputs:

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

## The staging process, idealized

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)
```

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
```

# Inner product

```
let dot
  : int → float array → float array → float
= fun n l r →
  let rec loop i =
    if i = n then 0.
    else l.(i) *. r.(i)
      +. loop (i + 1)
  in loop 0
```

## Inner product, loop unrolling

```
let dot'
  : int → float array code → float array code → float code
= fun n l r →
  let rec loop i =
    if i = n then .< 0. >.
    else .< ((.~l).(i) *. (.~r).(i))
            +. .~(loop (i + 1)) >.
  in loop 0
```

## Inner product, loop unrolling

```
# .< fun l r → .~(dot' 3 .<l>..<r>.) >;;  
- : (float array → float array → float) code =  
.< fun l r →  
    (l.(0) *. r.(0)) +.  
    ((l.(1) *. r.(1)) +. ((l.(2) *. r.(2)) +. 0.))>.
```



## Inner product, eliding no-ops

```
let dot''  
  : float array → float array code → float code =  
fun l r →  
  let n = Array.length l in  
  let rec loop i =  
    if i = n then .< 0. >.  
    else match l.(i) with  
      0.0 → loop (i + 1)  
    | 1.0 → .< (.~r).(i) +. .~(loop (i + 1)) >.  
    | x → .< (x *. (.~r).(i)) +. .~(loop (i + 1)) >.  
  in loop 0
```

## Inner product, eliding no-ops

```
# .< fun r → .~(dot'' [| 1.0; 0.0; 3.5 |] .<r>.) >.;;  
- : (float array → float) code =  
.< fun r → r.(0) +. ((3.5 *. r.(2)) +. 0.)>.
```

## Next: partially-static data; staged generic programming

```
type 'u genericQ = {D:DATA} → D.t code → 'u code
```