Staging

(March 2018)

Last time: generic programming

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

Review: abstraction

Lambda abstraction

λx:A.M
ΛA::K.M
λA::K.B
λx:A.B

Abstraction of type equalities

$\mathtt{a} \equiv \mathtt{b}$

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

```
type t = {f: 'a. ... }
type t = E: 'a s → t
```

Interfaces to computation

```
m >>= k        f ⊗ p
     effect E:t
```

Modular abstraction

```
module F(X : T) = ...
let f {X: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 (* indirect call *)

both_eq2 (fun x y → x = y)
```

```
type eq = { eq: 'a. 'a → 'a → bool }
let both_eq {eq} (x1, y1) (x2, y2) =
  (* indirect call through polymorphic function *)
  eq x1 x2 && eq y1 y2
```

The cost of ignorance: interpretative overhead

No interpretative overhead

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

Interpreting domain-specific values

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

Interpreting generic values

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

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)
```

```
val pow : int → int → int
```

`pow2`, `pow3`, `pow5` are more efficient, but pow is higher-level.

Can we combine high-level **abstraction** & low-level **performance**?

MetaOCaml basics

Quoting

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

 ⇝ "wx"
```

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

 ⇝ "x ^ y"
```

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

 ⇝ "ww"
```

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

 ⇝ "xw"
```

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 >.`	(values)
`.< 1 + 2 >.`	(expressions)
`.< [1; 2; 3] >.`	(structured values)
`.< x + y >.`	(expressions with free variables)
`.< fun x→ x >.`	(higher-order values)
`.< (.~f) 3 >.`	(splicing variables)
`.< .~(f 3) >.`	(splicing expressions)
`.< fun x→ .~(f .< x >.) >.`	(passing open code)

Quoting: typing

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

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

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

Quoting: open code

Open code (supports symbolic computation)

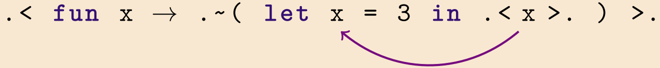
```
let pow_code n = .< fun x → .~(pow .< x >. n) >.
```

Cross-stage persistence

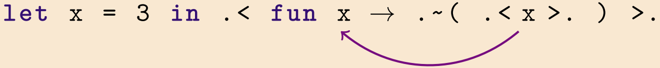
```
let print_int_pair (x,y) =
  Printf.printf "(%d,%d)" x y

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

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

Quoting: scoping

Scoping is **lexical**, just as in OCaml. Quotes do not affect scoping:



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



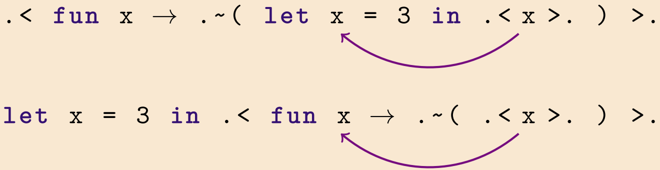
```
let x = 3 in .< fun x → .~( .< x >. ) >.
```

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. Quotes do not affect scoping:



```
.< 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: ?

```
.< 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: ?

```
.< 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: ?

```
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: ?

```
.< fun x → .~(!. .< x >. ) >.
```

Error: running open code

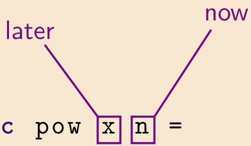
```
# .< fun x → .~(!. .< x >. ) >.;;
Exception:
Failure
 "The code built at Characters 7-8:\n
  .< fun x → .~(!. .< x >. ) >.;;\n
         ^\n
 is not closed: identifier x_2 bound at
 Characters 7-8:\n
  .< fun x → .~(!. .< x >. ) >.;;\n
         ^\n
 is free".
```

Learning by doing

Power, unstaged

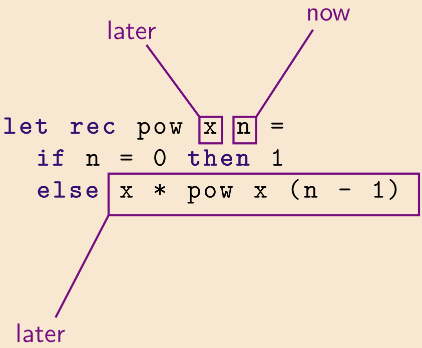
```
let rec pow x n =
  if n = 0 then 1
  else x * pow x (n - 1)
```

Power, unstaged



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

Power, unstaged



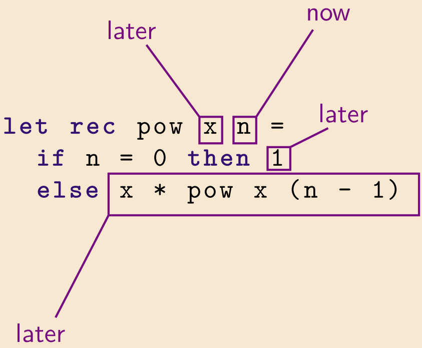
later

now

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

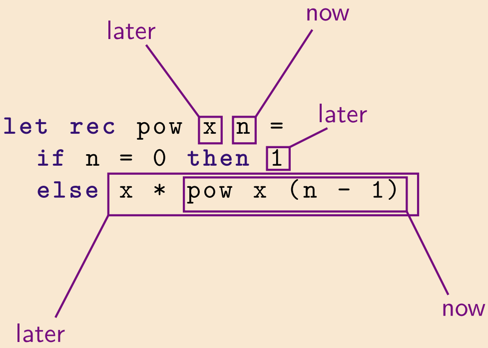
later

Power, unstaged



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

Power, unstaged



```
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)) >.
```

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
```

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

Specification:

$\texttt{dot n [|}x_1\texttt{; }x_2\texttt{; ...; }x_n\texttt{|] [|}y_1\texttt{; }y_2\texttt{; ...; }y_n\texttt{|]}$

$\rightsquigarrow (x_1 \times y_1) + (x_2 \times y_2) + \ldots + (x_n \times y_n)$

Implementation:

```
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
```

Binding-time analysis

Classify variables into **dynamic** (`'a code`) / **static** (`'a`)

```
let dot
 : int → float array code → float array code → float code
  = fun n l r →
```

dynamic: `l`, `r`
static: `n`

Classify expressions into static (no dynamic variables) / dynamic

```
   if i = n then 0
   else l.(i) *. r.(i)
```

dynamic: `l.(i) *. r.(i)`
static: `i = n`

Goal: reduce static expressions during code generation.

Inner product, loop unrolling

Length-specialized `dot`:

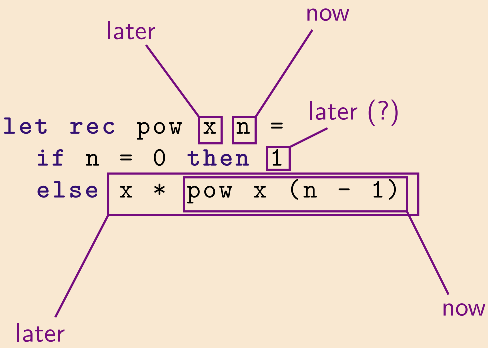
```
let dot'
: int → float array code → float array code → float code
 = fun n l r →
     let rec loop i =
       if i = n then .< 0. >.
       else .< ((.~l).(i) *. (.~r).(i))
                +. .~(loop (i + 1)) >.
     in loop 0
```

As with `pow`, making the loop variable static unrolls the loop:

```
# .< 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.))>.
```

Partially-static data

A closer look at `power`



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

Observation: `1` is misclassified: it's actually available **now**.
It's treated as dynamic (available later) so branch types match.

Problem: the static/dynamic distinction is too crude.

Sub-optimal code for `power`

The crudely-analysed `pow` generates **sub-optimal code**:

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

Generated code:

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

wasteful!

How should we fix `power`? (first attempt)

Solution one: rewrite pow to handle `n = 1`:

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

Generated code:

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

Objection: changing code **structure** to help staging is undesirable

How should we fix `power`? (second attempt)

Solution two: introduce a **type that subsumes static & dynamic**

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

and a function that **converts `sd` values to `code`**

```
let cd : 'a. 'a sd → 'a code = function
 | Sta s → .< s >.  (* (cross-stage persistence) *)
 | Dyn d → d
```

and **multiplication** for `sd` values that special-cases 1 and 0:

```
let (<*>) : int sd → int sd → int sd =
  fun x y → match x, y with
    Sta x, Sta y         → x * y
  | Sta 0, _ | _, Sta 0 → Sta 0
  | Sta 1, y | y, Sta 1 → y
  | x, y → .< .~(cd x) * .~(cd y) >.
```

Finally, **rewrite** `pow` to use `sd`:

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

How should we fix `power`? (second attempt: problems)

The `sd` type **fixes `pow`** (without changing code structure!)

However, `sd` is **not a complete solution**.

Consider the generated code for the following expression:

```
(Sta 2 <*> Dyn .<x>.) <*> Sta 3

  ⇝ .< 2 * x * 3 >.
```

We can simplify further (since $*$ is **associative** & **commutative**).

Plan: build $*$-specific representation that uses all the laws.

How should we fix power? (final attempt)

Monoid interface

```
module type MONOID = sig
 type t
 val unit : t
 val (<*>) : t → t → t
end
```

(Commutative) monoid laws

```
    unit <*> x ≡ x
    x <*> unit ≡ x
(x <*> y) <*> z ≡ x <*> (y <*> z)

       x <*> y ≡ y <*> x
```

Given a `MONOID` implementation `M`:

```
module type PS_monoid = sig
 type ps
 module N: MONOID with type t = ps
 val dyn: M.t code → ps
 val sta: M.t      → ps
 val eva: {O:MONOID} → (M.t code → O.t) → (M.t → O.t) →
   ps → O.t
end
```

How should we fix power? (final attempt)

Monoid interface

```
module type MONOID = sig
  type t
  val unit : t
  val (<*>) : t → t → t
end
```

(Commutative) monoid laws

```
    unit <*> x ≡ x
    x <*> unit ≡ x
(x <*> y) <*> z ≡ x <*> (y <*> z)

        x <*> y ≡ y <*> x
```

Given a `MONOID` implementation `M`:

type of partially-static values

```
module type PS_monoid = sig
  type ps
  module N: MONOID with type t = ps
  val dyn: M.t code → ps
  val sta: M.t      → ps
  val eva: {O:MONOID} → (M.t code → O.t) → (M.t → O.t) →
    ps → O.t
end
```

How should we fix power? (final attempt)

Monoid interface

```
module type MONOID = sig
 type t
 val unit : t
 val (<*>) : t → t → t
end
```

(Commutative) monoid laws

```
     unit <*> x ≡ x
     x <*> unit ≡ x
(x <*> y) <*> z ≡ x <*> (y <*> z)

          x <*> y ≡ y <*> x
```

Given a `MONOID` implementation `M`:

type of partially-static values

ps is a MONOID

```
module type PS_monoid = sig
 type ps
 module N: MONOID with type t = ps
 val dyn: M.t code → ps
 val sta: M.t      → ps
 val eva: {O:MONOID} → (M.t code → O.t) → (M.t → O.t) →
   ps → O.t
end
```

How should we fix power? (final attempt)

Monoid interface

```
module type MONOID = sig
 type t
 val unit : t
 val (<*>) : t → t → t
end
```

(Commutative) monoid laws

```
    unit <*> x ≡ x
    x <*> unit ≡ x
(x <*> y) <*> z ≡ x <*> (y <*> z)

      x <*> y ≡ y <*> x
```

Given a `MONOID` implementation `M`:

```
module type PS_monoid = sig
 type ps
 module N: MONOID with type t = ps
 val dyn: M.t code → ps
 val sta: M.t      → ps
 val eva: {O:MONOID} → (M.t code → O.t) → (M.t → O.t) →
   ps → O.t
end
```

type of partially-static values

ps is a MONOID

static/dynamic injections

How should we fix power? (final attempt)

Monoid interface

```
module type MONOID = sig
 type t
 val unit : t
 val (<*>) : t → t → t
end
```

(Commutative) monoid laws

```
   unit <*> x ≡ x
   x <*> unit ≡ x
(x <*> y) <*> z ≡ x <*> (y <*> z)

       x <*> y ≡ y <*> x
```

Given a `MONOID` implementation `M`:

```
module type PSmonoid = sig
 type ps
 module N: MONOID with type t = ps
 val dyn: M.t code → ps
 val sta: M.t      → ps
 val eva: {O:MONOID} → (M.t code → O.t) → (M.t → O.t) →
   ps → O.t
end
```

type of partially-static values

ps is a MONOID

static/dynamic injections

elimination into other monoids

How should we fix `power`? (final attempt)

So far $\texttt{PS}_{\texttt{monoid}}$ looks a lot like `sd`.

But the `sd` implementation does not make full use of `MONOID` laws:

```
(dyn .<x>. <*> sta 2) <*> sta 4   ≡   dyn .<x>. <*> (sta 2 <*> sta 4)
                                ⇓              ⇓
            Dyn .<(x * 2) * 4>    ≡   Dyn .<x * 8>.
```

Law-observing partially-static commutative monoid (sketch)

Representation: static value & bag of variables ($sx^ny^m\ldots$)

```
type 'a var (* dynamic variables *)
module IVarMap : Map.S with type key = int var
type ps_cmonoid = int * int IVarMap.t
```

How should we fix power? (final attempt)

Law-observing partially-static commutative monoid (sketch)

Representation: static value & bag of variables ($sx^ny^m\ldots$)

```
type 'a var (* dynamic variables *)
module IVarMap : Map.S with type key = int var
type ps_cmonoid = int * int IVarMap.t
```

Evaluation produces a **canonical** representation

```
   (sta 2 <*> dyn .<x>.) <*> (sta 4 <*> dyn .<x>.)
⇝ (2, {x↦1}) <*> (4, {x↦1})
⇝ (8, {x↦2})
```

Code generation minimizes the number of **operations**

```
   cd (pow .<x>. 5)
⇝ cd (1, {x↦5})
⇝ .< let y = x*x in let z = y*y in z*x >.
```

(Exercise (non-trivial): write this improved cd using eva.)

Next time: more staging

`.<e>.`

Generalizing **algebraic** optmisation

Staging and **effects**