Last time: staging basics

.< e >.

Staging recap

Goal: specialise with available data to improve future performance

New constructs: `'a code` `.< e >.` `.~e` `!.e`

Example: pow

Improvements: unrolling loops

Power, staged

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

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

```
# 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

```
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.)>.
```

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

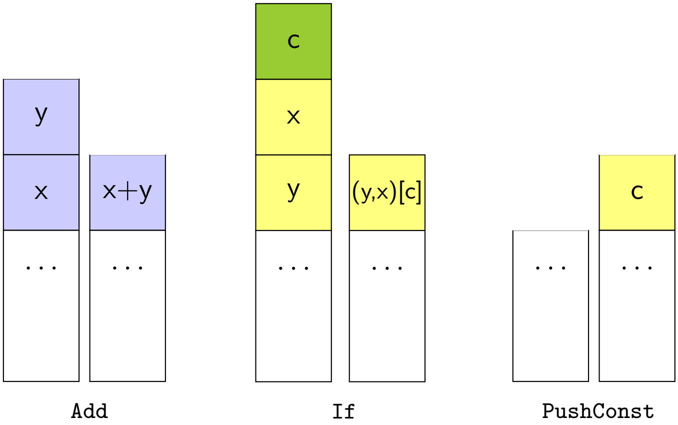
```
if i = n then 0
else l.(i) *. r.(i)
```

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

Goal: reduce static expressions during code generation.

Partially-static data structures

Stack machines again



Stack machines: higher-order vs first-order

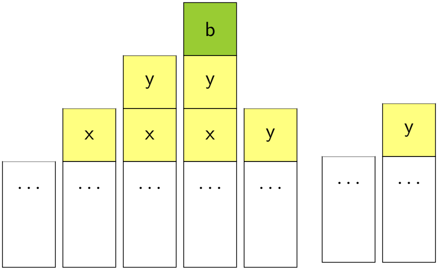
```
type ('s, 't) t = 's → 't
let add (x, (y, s)) = (x + y, s)
```

```
type ('s, 't) t = ('s, 't) instrs
let add = Add :: Stop
```

Recap: optimising stack machines

```
val (>>=) : 'a t → ('a → 'b t) → 'b t

val (⊗) : ('a → 'b) t → 'a t → 'b t
```




```
PushConst x
PushConst y
PushConst true
If
```

```
PushConst
    y
```

Stack machines: basic interface

```
module type STACKM =
sig
  type ('s, 't) t
  val nothing : ('s, 's) t
  val (⊗) : ('r, 's) t →
            ('s, 't) t →
            ('r, 't) t
  val add : (int * (int * 's),
                   int * 's) t
  val _if_ : (bool * ('a * ('a * 's)),
                              'a * 's) t
  val push_const : 'a → ('s,
                    'a * 's) t
  val execute : ('s, 't) t → 's → 't
end
```

Higher-order stack machines

```
module StackM : STACKM =
struct
 type ('s, 't) t = 's → 't
 let nothing s = s
 let (⊗) f x s = x (f s)
 let add (x, (y, s)) = ((x + y, s))
 let _if_ (c, (x, (y, s))) = ((if c then x else y), s)
 let push_const v s = (v, s)
 let execute f s = f s
end
```

Optimising higher-order stack machines

Why are the higher-order machines hard to optimize?

```
let (⊗) f x s = x (f s)
let push_const v s = (v, s)
let add (x, (y, s)) = ((x + y, s))
```

```
push_const 3 ⊗
push_const 4 ⊗
add
```

Optimising higher-order stack machines

Why are the higher-order machines hard to optimize?

```
let (⊗) f x s = x (f s)
let push_const v s = (v, s)
let add (x, (y, s)) = ((x + y, s))
```

Inlining `push_const`, `add`:

```
(fun s → (3, s)) ⊗
(fun s → (4, s)) ⊗
(fun (x, (y, s)) → ((x + y, s)))
```

Optimising higher-order stack machines

Why are the higher-order machines hard to optimize?

```
let (⊗) f x s = x (f s)
let push_const v s = (v, s)
let add (x, (y, s)) = ((x + y, s))
```

Inlining ⊗:

```
(fun s →
   (fun (x, (y, s)) → ((x + y, s)))
   ((fun s → (fun s → (4, s)) ((fun s → (3, s)) s))
       s))
```

Optimising higher-order stack machines

Why are the higher-order machines hard to optimize?

```
let (⊗) f x s = x (f s)
let push_const v s = (v, s)
let add (x, (y, s)) = ((x + y, s))
```

Inlining ⊗:

```
(fun s →
   (fun (x, (y, s)) → ((x + y, s)))
   ((fun s → (fun s → (4, s)) ((fun s → (3, s)) s))
      s))
```

Difficulty: **evaluating under lambda**

Stack machines: higher-order vs first-order vs staged

```
type ('s, 't) t = 's → 't
let add (x, (y, s)) = (x + y, s)
```

```
type ('s, 't) t = ('s, 't) instrs
let add = Add :: Stop
```

```
type ('s, 't) t = 's code → 't code
let add p = .<let (x, (y, s)) = .~p in (x + y, s)>.
```

Staging the higher-order stack machine

```
module type STACKM_staged =
sig
  include STACKM
  val compile : ('s, 't) t → ('s → 't) code
end
```

Staging the higher-order stack machine

```
module StackM_staged : STACKM_staged =
struct
  type ('s, 't) t = 's code → 't code
  let nothing s = s
  let (⊗) f x s = x (f s)
  let add p =
    (.< let (x, (y, s)) = .~p in
       (x + y, s) >.)
  let _if_ p =
   .< let (c, (x, (y, s))) = .~p in
        ((if c then x else y), s) >.
  let push_const v s =
   .< (v, .~s) >.

  let compile f = .< fun s → .~(f .<s>.) >.
  let execute f s = !.(compile f) s
end
```

Staging the higher-order stack machine: output

```
# compile (push_const true ⊗ _if_);;
- : ('_a * ('_a * '_b) → '_a * '_b) code =
.< fun s_59 →
     let (c,(x,(y,s))) = (true, s) in
     ((if c then x else y), s)>.

# compile (push_const 3 ⊗ push_const 4 ⊗
           push_const false ⊗ _if_);;
  - : ('_a → int * '_a) code =
.< fun s →
     let (c,(x,(y,s))) = (false, (4, (3, s))) in
     ((if c then x else y), s)>.

# compile (push_const 3 ⊗ push_const 4 ⊗
           push_const false ⊗ _if_);;
  - : ('_a → int * '_a) code =
.< fun s →
     let (c,(x,(y,s))) = (false, (4, (3, s))) in
     ((if c then x else y), s)>.
```

Possibly-static values

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

let unsd : 'a.'a sd → 'a code =
 function
    Sta v →.<v>.
  | Dyn v → v
```

Partially-static stacks

```
type 'a stack =
    Tail : 'a code → 'a stack
  | :: : 'a sd * 'b stack → ('a * 'b) stack

let rec unsd_stack : type s.s stack → s code =
  function
    Tail s → s
  | c :: s → .<(.~(unsd c), .~(unsd_stack s)) >.
```

Stack machine: binding-time analysis

```
type ('s, 't) t = 's → 't
let add (x, (y, s)) = (x + y, s)
```

```
type ('s, 't) t = ('s, 't) instrs
let add = Add :: Stop
```

```
type ('s, 't) t = 's code → 't code
let add p = .<let (x, (y, s)) = .~p in (x + y, s)>.
```

```
type ('s, 't) t = 's stack → 't stack
let rec add : type s.(int * (int * s), int * s) t =
 function
    Sta x :: Sta y :: s → Sta (x + y) :: s
  | ...
```

Stack machine: optimising add

```
let extend : 'a 'b.('a * 'b) stack → ('a * 'b) stack =
 function
    Tail s → Dyn .<fst .~s >. :: Tail .< snd .~s >.
  | _ :: _ as s → s

let rec add : type s.(int * (int * s), int * s) t =
 function
    Sta x :: Sta y :: s → Sta (x + y) :: s
  | x :: y :: s → Dyn .<.~(unsd x) + .~(unsd y) >. :: s
  | Tail _ as s → add (extend s)
  | c :: (Tail _ as s) → add (c :: extend s)
```

Stack machine: optimising branches

```
let rec _if_
 : type s a.(bool * (a * (a * s)), a * s) t =
 function
 | Sta true :: x :: y :: s → x :: s
 | Sta false :: x :: y :: s → y :: s
 | Dyn c :: x :: y :: s →
   Dyn .< if .~c then .~(unsd y) else .~(unsd x) >.  :: s
 | (Tail _ as s) → _if_ (extend s)
 | c :: (Tail _ as s) → _if_ (c :: extend s)
 | c :: x :: (Tail _ as s) →
   _if_ (c :: x :: extend s)
```

Stack machine: top-level compilation

```
val compile : ('s, 't) t → ('s → 't) code

let compile f =
  .< fun s → .~(unsd_stack (f (Tail .<s>.)) ) >.
```

Stack machine: flexible optimisation

```
# compile add;;
- : (int * (int * '_a) → int * '_a) code =
.< fun s → ((fst s + fst (snd s)), snd (snd s))>.

# compile _if_;;
- : (bool * ('_a * ('_a * '_b)) → '_a * '_b) code =
 .< fun s →
      ((if fst s
        then fst (snd (snd s))
        else fst (snd s)),
        (snd (snd (snd s))))>.

# compile (push_const true ⊗ _if_);;
- : ('_a * ('_a * '_b) → '_a * '_b) code =
 .< fun s → (fst s, snd (snd s))>.
```

Stack machine: flexible optimisation

```
# compile (push_const false ⊗ _if_);;
- : ('_a * ('_a * '_b) → '_a * '_b) code =
.< fun s → (fst (snd s), snd (snd s))>.

# compile (push_const 3 ⊗ push_const 4 ⊗
           push_const false ⊗ _if_);;
- : ('_a → int * '_a) code =
.< fun s → (3, s)>.

# compile (push_const 3 ⊗ push_const 4 ⊗
           add ⊗ push_const 2 ⊗
           push_const false ⊗ _if_);;
- : ('_a → int * '_a) code =
.< fun s → (7, s)>.
```

Staging generic programming

```
val gshow : 'a data → ('a → string) code
```

Generic programming: binding-time analysis

```
gshow.q (list (int * bool)) [(1, true); (2; false)]
```

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

We've used type representations to traverse values.

Now we'll use type representations to generate code.

Goal: generate code that contains no `typeable` or `data` values.

Desired code for gshow

```
val gshow : 'a data → ('a → string) code
```

```
type tree =
   Empty : tree
 | Branch : branch → tree
and branch = tree * int * tree
```

```
let rec show_tree = function
  Empty → "Empty"
| Branch b → "(Branch "^ show_branch b ^")"
and show_branch (l, v, r) =
  show_tree l ^", "^ show_int v ^", "^ show_tree r
```

Generic programming

Type equality

```
type 'a typeable

val int : int typeable

val (=~=) :
  'a typeable → 'b typeable → ('a,'b) eql option
```

Traversals

```
type 'a data
and 'u genericQ =
  { q: 't. 't data → 't → 'u }

val int : int data

val gmapQ : 'u genericQ → 'u list genericQ
```

Generic functions

```
val gshow : string genericQ
```

Generic programming, staged

Type equality

```
type 'a typeable

val int : int typeable

val (=~=) :
  'a typeable → 'b typeable → ('a,'b) eql option
```

Traversals

```
type 'a data
and 'u genericQ =
  { q: 't. 't data → 't code → 'u code }

val int : int data

val gmapQ : 'u genericQ → 'u list genericQ
```

Generic functions

```
val gshow : string genericQ
```

Staging gmapQ

```
let ( * ) a b = {
  ...
  gmapQ = fun { q } (x, y) → [q a x; q b y];
}
```

```
let ( * ) a b = {
  ...
  gmapQ = fun { q } p →
   .< let (x, y) = .~p in [.~(q a .<x>.); .~(q b .<y>.)]>.
}
```

Staging gshow

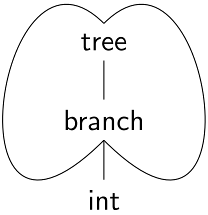
```
let rec gshow : string genericQ =
 { q =
    fun data v ->
      "("^ data.constructor v
      ^ String.concat " " ((gmapQ gshow).q data v)
      ^ ")" }
```

Difficulty: **recursion**

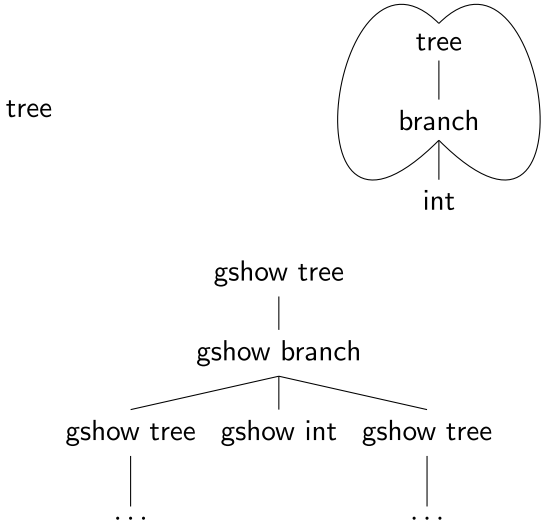
Cyclic static structures

Cyclic type structures

tree



Cyclic type structures



Memoization

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

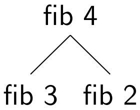
Memoization

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

fib 4

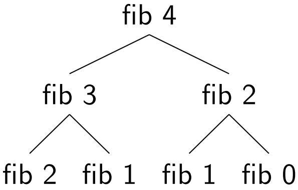
Memoization

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



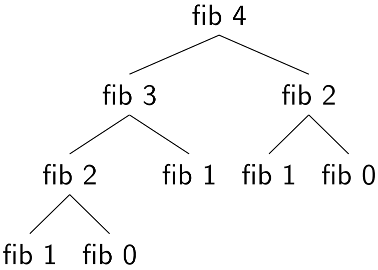
Memoization

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



Memoization

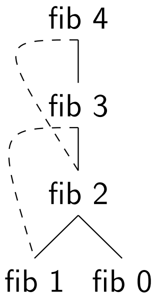
```
let rec fib = function
    0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```



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
```

Typed maps

```
type t
val empty : t
val add : t → 'a data → ('a → string) code → t
val lookup : t → 'a data → ('a → string) code option
```

Typed maps

```
type t
val empty : t
val add : t → 'a data → ('a → string) code → t
val lookup : t → 'a data → ('a → string) code option
```

```
type t =
    Nil : t
  | Cons : 'a data * ('a → string) code * t → t
```

Typed maps

```
type t
val empty : t
val add : t → 'a data → ('a → string) code → t
val lookup : t → 'a data → ('a → string) code option
```

```
type t =
    Nil : t
  | Cons : 'a data * ('a → string) code * t → t

let empty = Nil
let add t d x = Cons (d, x, t)
let rec lookup :
  type a.t → a data → (a → string) code option =
  fun t l → match t with
      Nil → None
    | Cons (r, d, rest) →
      match l.typeable =~= r.typeable with
        Some Refl → Some d
      | None → lookup rest l
```

Generating recursive definitions

Mutually-recursive definitions

```
let rec evenp x =
  x = 0 || oddp (pred x)
and oddp x =
  not (evenp x)
```

Difficulty: building up arbitrary-size `let rec` ... `and` ... `and` ....

n-ary operators are difficult to abstract!

Recursion via mutable state

```
let evenp = ref (fun _ → assert false)
let oddp  = ref (fun _ → assert false)

evenp := fun x → x = 0 || !oddp (pred x)
oddp := fun x → not (!evenp x)
```

What if `evenp` and `oddp` generated in different parts of the code?

Plan: use `let`-insertion to interleave bindings and assignments.

Let insertion

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

```
.< 1 +
     .~(let_locus (fun () →
        .< 2 + .~(genlet .< 3 + 4 >.) >.)) >.
```

```
1 +
  let x = 3 + 4 in
  2 + x
```

Let rec insertion

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

```
let letrec k =
  let r = genlet (.< ref (fun _ -> assert false) >.) in
  let _ = genlet (.<.~r := .~(k .< ! .~r >.) >.) in
 .< ! .~r >.
```

Generating code for gshow

```
val memofix : (string genericQ → string genericQ) →
              string genericQ
```

```
let memofix h =
{ q = fun t →
  let tbl = ref empty in
  let rec result d x = match lookup !tbl d with
       Some f → .<.~f .~x >.
     | None →
        let g = letrec (fun self ->
                  tbl := add !tbl d self;
                 .< fun y -> .~(h result .<y>.) >.)
         in .< .~g .~x >.
```

Generating code for gshow

```
val memofix : (string genericQ → string genericQ) →
              string genericQ


let gshow_gen : string genericQ → string genericQ =
    fun gshow →
      { q = fun data v →
              .< "("^ .~(data.constructor v)
                ^ String.concat " " .~((gmapQ gshow).q
                   data v)
                ^ ")" >.}

let gshow = memofix gshow_gen
```

Generated code for gshow

```
let show_tree = ref (fun _ → assert false) in
let show_branch = ref (fun _ → assert false) in
let show_int = ref (fun _ → assert false) 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
```

Next time: reagents