Last time: staging basics

`.< e >.`

Staging `pow`

```
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
```

A second example: 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
```

A second example: 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
```

Question: how can we specialize `dot` to improve performance?

Inner product: loop unrolling

Given the **length** in advance, we can unroll the loop:

```
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
```

Unrolling in action

```
# .< 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

Given one **vector** in advance, we can simplify the arithmetic:

```
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
```

Simplification in action

```
# .< 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

Possibly-static data

Observation: data may not be entirely static or entirely dynamic

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

Problem: naive binding-time analysis turns everything dynamic

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

Solution: *possibly-static data*

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

Result: finer-grained classification, preserving staticness

```
if i = n then Sta 0
else Dyn .< .~l.(i) *. .~r.(i) >.
```

Dynamizing possibly-static data

Possibly-static data can be made fully dynamic:

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

Possibly-static integers

```
module type NUM = sig
  type t
  val (+) : t → t → t
  ...
end
```

```
implicit module Num_int_sd: NUM with type t = int sd =
struct
  type t = int sd
  let (+) l r = match l, r with
   | Sta 0, v
   | v, Sta 0 → v
   | Sta l, Sta r → Sta (l + r)
   | l, r → Dyn .<.~(cd l) + .~(cd r) >.
end
```

```
Sta 2 + Sta 3          ⇝   Sta 5
Sta 0 + Dyn .< x >.    ⇝   Dyn .< x >.
Dyn .< x >. + Dyn .< y >.    ⇝   Dyn .< x + y >.
```

`dot` with possibly-static elements

`dot` with overloading, without staging

```
let dot: {N:NUM} → int → N.t array → N.t array → N.t
 = fun {N:NUM} n l r →
    let rec loop i =
      if i = n then N.zero
      else l.(i) * r.(i)
        + loop (i + 1)
    in loop 0
```

`dot` instantiated with `Num_int_sd`:

```
# dot 3 [|Sta 1;     Sta 0;     Dyn .< 3 >.|]
        [|Dyn .< 2 >.; Dyn .< 1 >.; Sta 0    |]
- : int sd =
Dyn .< 2 >.
```

Partially-static data

Problem: possibly-static data is still too coarse

```
Sta 2 + Dyn .<x>. + Sta 3
⤳
Dyn .<2 + x + 3>.
```

Solution: maintain more structure using **partially-static** data

Examples:

trees with static shapes and dynamic labels
lists with static prefixes and dynamic tails
products with one static and one dynamic element
. . . many more!

Partially-static integers

```
type ps_int = { sta : int;
                dyn : int code list }

implicit module Num_ps_int: NUM with type t = ps_int =
struct
  type t = ps_int
  let (+) l r =
     { sta = l.sta + r.sta; dyn = l.dyn @ r.dyn }
end

let dyn { sta; dyn } =
   fold_left (fun x y →.< .~x + .~y >.) .< sta >. dyn

let sta x = {sta=x; dyn=[]}
let dyn x = {sta=0; dyn=[x]}

cd (sta 2 + dyn .< x >. + sta 3)
⇝
.< x + 5 >.
```

`let` insertion

`let` insertion: motivation

Problem: inserting generated code in place is not always optimal

Example: the code built by `f` may not depend on `i`:

```
let generate_loop f =
  .< fun e →
    for i = 0 to 10 do print .~( f .<e>. .<i>. ) done >.
```

```
generate_loop (fun e →.< .~e ^ "\n" >.)
⇝
  .< fun e →
    for i = 0 to 10 do
      print (e ^ "\n")    (* repeated work! *)
    done >.
```

What we need: A way to insert `let` bindings at outer levels

```
  .< fun e →
       let c = e ^ "\n" in
       for i = 0 to 10 do print c done >.
```

`let` insertion: a simple implementation

`let` insertion as an effect

```
effect GenLet : 'a code → 'a code
```

```
let genlet v = perform (GenLet v)
```

Handling `let` insertion

```
let let_locus : (unit → 'a code) → 'a code =
  fun f → match f () with
  | x → x
  | effect (GenLet e) k →
   .< let x = .~e in .~(continue k .< x >.)>.
```

`let` insertion in action

Example

```
let_locus
 (fun () →
  .< w + .~(genlet .< y + z >.) >.)
```

Captured continuation

```
.< w + .~( - ) >.
```

`let` generation

```
| effect (GenLet e) k →
 .< let x = .~e in .~(continue k .< x >.)>.
```

Result

```
.< let x = y + z in
      w + x >.
```

Where to insert `let`?

Sometimes there are several possible insertion points for `let`

For example, consider the following program:

```
.< fun y → y + .~(genlet e) >.
```

We could insert `let` *beneath* the binding for `y`

```
.< fun y → let x = .~e in y + x >.
```

Or *above*:

```
.< let x = .~e in fun y → y + x >.
```

We typically want the **highest point where `e` is well-scoped**.

`let` insertion at the outermost valid point

Is `e` well-scoped at this point in the program?

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

`genlet` defaults to insertion-in-place

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

`let_locus` searches the stack for the highest suitable handler

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

`let rec` insertion

Question: how can we generate (mutually) recursive functions?

```
let rec evenp x = x = 0 || oddp (x - 1)
    and oddp  x = not (evenp x)
```

Difficulty: constructing binding groups of unknown size

Observation: *n*-ary operators are difficult to abstract!

Recursion via references (Landin's knot)

```
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 rec` insertion with references

`letrec` via `genlet`

```
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 >.
```

`letrec` in action

```
let fib = let_locus @@ fun () →
  letrec (fun f →
        .< fun x → if x = 0 then 1 else x * .~f (x - 1) >.)
⇝
.<let r = ref (fun _  → assert false) in
  let _ = r := (fun x →if x = 0 then 1 else x * !r (x - 1))
   in !r>.
```

Staging generic programming

Generic programming recap

Type equality

```
val eqty : {A:TYPEABLE} → {B:TYPEABLE} →
   (A.t, B.t) eq option
```

Generic shallow traversals

```
type 'u genericQ = {D:DATA} → D.t → 'u
val gmapQ : 'u genericQ → 'u list genericQ
```

Generic recursive schemes

```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v ^ concat " " (gmapQ gshow v) ^ ")"
```

`gshow` in action

```
gshow [1;2;3]   ⇝ "(1 :: (2 :: (3 :: ([]))))"
```

Generic programming vs hand-written code

Generic `show`

```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v ^ concat " " (gmapQ gshow v) ^ ")"
```

Hand-written `show`

```
let rec show_list: ('a→string)→'a list→string =
 fun f l →
   match l with
   | [] → "[]"
   | h::t → "("^ f h ^" :: "^ show_list f t ^")"
```

Generic programming vs hand-written code

Generic `show`

```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v ^ concat " " (gmapQ gshow v) ^ ")"
```

Hand-written `show`

```
let rec show_list: ('a→string)→'a list→string =
 fun f l →
   match l with
   | [] → "[]"
   | h::t → "("^ f h ^" :: "^ show_list f t ^")"
```

Performance difference: an order of magnitude

Generic programming vs hand-written code

Generic `show`

```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v ^ concat " " (gmapQ gshow v) ^ ")"
```

Hand-written `show`

```
let rec show_list: ('a→string)→'a list→string =
 fun f l →
   match l with
   | [] → "[]"
   | h::t → "("^ f h ^" :: "^ show_list f t ^")"
```

Performance difference: an order of magnitude

Plan: turn `gshow` **into a code generator**

Generic programming: binding-time analysis

```
gshow {Data_list{Data_int}} [1; 2; 3]
```

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.

Generic programming, staged

Type equality (unchanged)

```
val eqty : {A:TYPEABLE} → {B:TYPEABLE} →
   (A.t, B.t) eq option
```

Generic shallow traversals

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

Generic recursive schemes

```
 let gshow = gfixQ_ (fun self {D:DATA} v →
.< "("^ .~(constructor_ v)
       ^ concat " " .~(gmapQ_ self v) ^")" >.)
```

`gshow` in action

```
instantiate gshow   ⇝ .< let rec show = ... >.
```

Staging `gmapQ`

The type of staged `gmapQ`

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

Implementing staged `gmapQ`

```
implicit module rec DATA_list {A:DATA}
  : DATA with type t = A.t list =
struct
  let gmapQ q l =
 .< match .~l with
      | [] → []
      | h :: t → [.~(q .< h >.); .~(q .< t >.)] >.
  (* ... *)
end
```

Fixpoint operators

Problem: we can't overload / redefine `let rec`

```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v
    ^ concat " " (gmapQ gshow v) ^ ")"
```

Solution: rewrite gshow using a **fixpoint combinator**

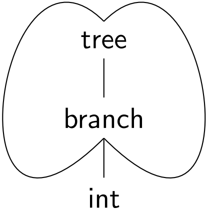
```
let rec gfixQ :
    (u genericQ → u genericQ) → u genericQ =
  fun f {D:DATA} x → f {D} (gfixQ f) x

let gshow = gfixQ (fun self {D:DATA} v →
  "("^ constructor_ v
    ^ concat " " (gmapQ self v) ^ ")")
```

New problem: stage `gfixQ`

`gfixQ`: cyclic static structures

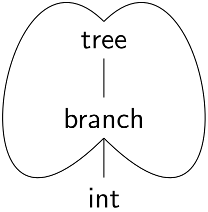
```
type tree =
   Empty : tree
 | Branch : branch → tree
and branch = tree * int * tree
```

tree



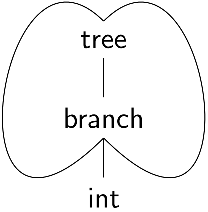
`gfixQ`: cyclic static structures

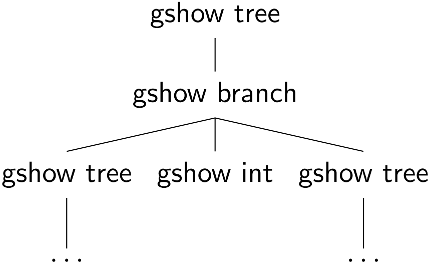
tree



`gfixQ`: cyclic static structures

tree





Background: memoization

Recursive functions can be inefficient

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

Background: memoization

Recursive functions can be inefficient

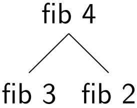
```
let rec fib = function
    0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```

fib 4

Background: memoization

Recursive functions can be inefficient

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



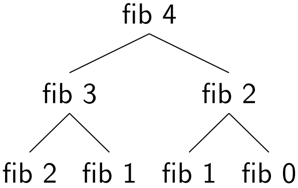
fib 4

fib 3 fib 2

Background: memoization

Recursive functions can be inefficient

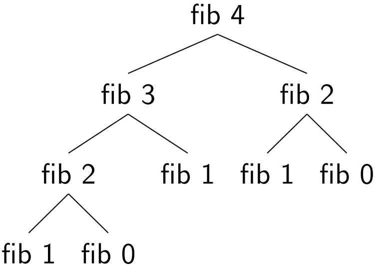
```
let rec fib = function
      0 → 0
    | 1 → 1
    | n → fib (n - 1) + fib (n - 2)
```



Background: memoization

Recursive functions can be inefficient

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



Background: memoization

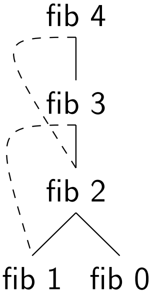
Recursive functions can be inefficient — use memoization

```
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
```

Background: memoization

Recursive functions can be inefficient — use memoization

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

let fib = memoize open_fib
```



Memoizing generic functions

A lookup table for heterogeneous code values

```
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
```

A staged generic fixpoint operator

```
let gfixQ (f : 'v genericQ → 'v genericQ) =
 let tbl = empty () in
 let rec result {D: DATA} x =
   match lookup !tbl with
   | Some g →.< .~g .~x >.
   | None → let g = letrec
                      (fun self →
                        add tbl self;
                      .< fun y → .~(f result .<y>.) >.)
              in .< .~g .~x >.
 in result
```

Staged `gshow`

`gshow`, **staged**

```
 let gshow = gfixQ_ (fun self {D:DATA} v →
.< "("^ .~(constructor_ v)
         ^ concat " " .~(gmapQ_ self v) ^")" >.)
```

Generated code for `gshow`

```
let show_list = 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_list :=
  (fun t →
  "("^((match t with [] → "[]"
                     | _ :: __ → "::") ^
       ((concat " "
          (match t with
           | [] → []
           | h :: t → [!show_int h;
                        !show_list t])) ^")"))) in
!show_list
```

Staging generic programming: summary

Bad news: the generated code is pretty poor

Better news: the performance is fairly good!

typically 10× the speed of generic code;
typically 0.5-1× the speed of handwritten code

Best news: the staging can be improved

with better `let` / `let rec` insertion
with partially-static data
with `match` insertion
with `match` elimination
. . . and many other such techniques

until it is as fast as handwritten code (& sometimes faster!)

Next time: super-advanced functional programming

```
data Vec (A : Set) : ℕ → Set
```