Last time: staging basics

$.\!< e >\!.$

Staging recap

Goal: specialise with available data to improve future performance

New constructs: 'a code .< e >. !. e .< x >.

Binding-time analysis: what is available statically?

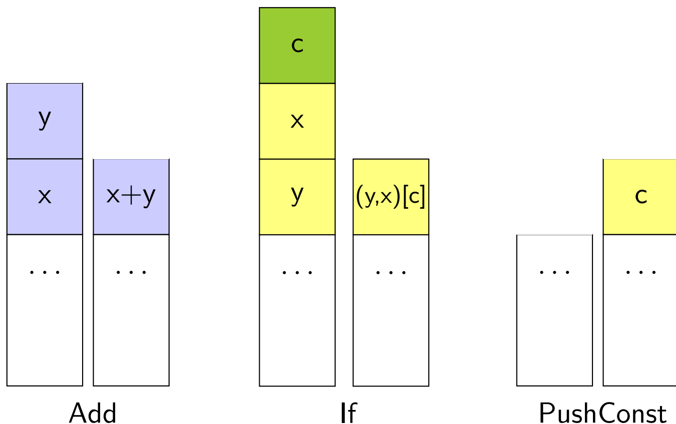
Idealized staging process: annotate, close, apply.

Examples: pow, dot

Improvements: unrolling loops, eliminating unnecessary work

Partially-static data structures

Stack machines again



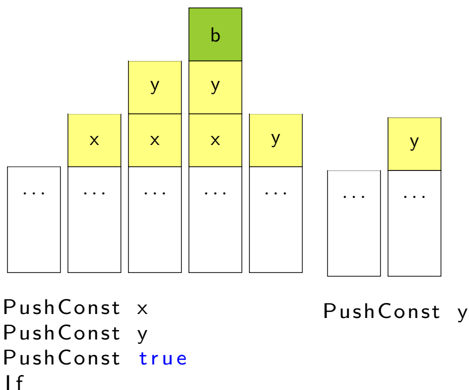
Stack machines: higher-order vs first-order

```
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
```



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)>.
```

Staged generic programming

val gshow : 'a data $\rightarrow$ ('a $\rightarrow$ string) code

Generic programming: binding-time analysis

```
gshow (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 without any type equality tests.

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

val int : int data

val gmapQ :
 'a data → (∀'b.'b data → 'b → 'u) →
 'a → 'u list
```

Generic functions

```
val gshow : 'a data → 'a → string
```

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

val int : int data

val gmapQ :
 'a data → (∀'b.'b data → 'b code → 'u code) →
 'a code → 'u list code
```

Generic functions

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

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

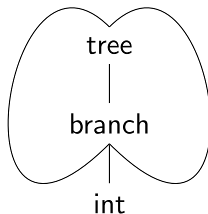
```
let rec gshow : 'a. 'a data → 'a → string =
  fun t v →
    "(" ^ t.constructor v
    ^ String.concat " " (t.gmapQ {q = gshow} v)
    ^ ")"
```

Difficulty: **recursion**

Cyclic static structures

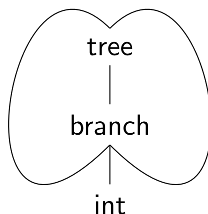
Cyclic type structures

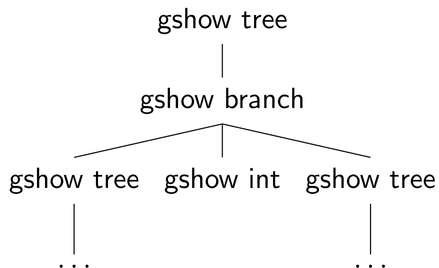
tree



Cyclic type structures

tree





Memoization

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

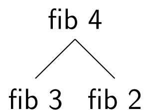
Memoization

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

fib 4

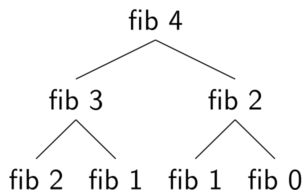
Memoization

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



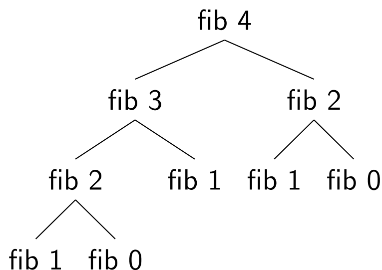
Memoization

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



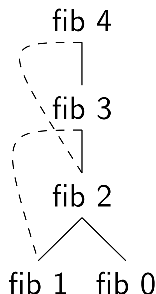
Memoization

```
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)
```

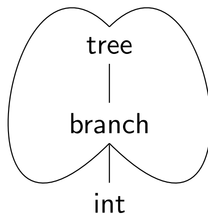


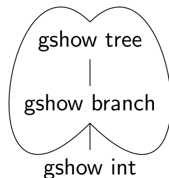
Memoization, factored

```
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
```

Memoizing functions over cyclic type structures

tree





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 → unit code) → 'c) → 'c

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

let_locus @@ fun () →
letrec (fun evenp def_evenp →
letrec (fun oddp def_oddp →
 def_evenp .< fun x → x = 0 || (.~oddp) (x − 1) >.;
 def_oddp .< fun x → x = 1 || not (.~evenp x) >.;
.< (.~evenp, .~oddp) >. ))
```

Let rec insertion

```
let_locus @@ fun () →
letrec (fun evenp def_evenp →
letrec (fun oddp def_oddp →
 def_evenp .< fun x → x = 0 || (.~oddp) (x − 1) >.;
 def_oddp .< fun x → x = 1 || not (.~evenp x) >.;
.< (.~evenp, .~oddp) >. ))
```

```
.<let e = ref (fun _ → assert false) in
  let o = ref (fun _ → assert false) in
  let _ = e := (fun x → x = 0 || !o (x − 1)) in
  let _ = o := (fun y → y = 1 || not (!e y)) in
    (!e, !o)>.
```

Generating code for gshow

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

```
let memofix h =
{ q = fun t →
  let tbl = ref empty in
  let rec h' : 'a. 'a data → 'a code → string code =
    fun d x → match lookup !tbl d with
        Some f → .<.~f .~x >.
      | None →
        letrec (fun f deff →
         tbl := add !tbl d f;
         let _ = deff .< fun y → .~((h {q=h'}).q d .<y>.) >.
          in .< .~f .~x >.)
  in fun x → let_locus @@ fun () → h' t x }
```

Generating code for gshow

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

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

let gshow x = (memofix gshow_gen).q x
```

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 t_12 = 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
```

Extension: better code generation

gshow currently generates code like this:

```
String.concat " "
 (match y with
    [] → []
  | x::xs → [!t₁ x; !t₂ xs])
```

How can we avoid generating intermediate lists?

```
match y with
  [] → ""
| x::xs → !t₁ x ^" "^ !t₂ xs
```

Extensions: trimming branches

Collecting bools

```
everything (list (int * bool))
  (@) (mkQ Typeable.bool [] (fun x → [x]))
```

Extensions: trimming branches

Collecting bools

```
everything (list (int * bool))
  (@) (mkQ Typeable.bool [] (fun x → [x]))
```

```
let rec find_bools_list = function
  [] → []
| (i,b) :: ps → find_bools_int i
              @ find_bools_bool b
              @ find_bools_list ps
and find_bools_int _ = []
and find_bools_bool b = [b]
```

Extensions: trimming branches

Collecting bools

```
everything (bool * list int)
  (@) (mkQ Typeable.bool [] (fun x → [x]))
```

```
let rec find_bools_pair (b, l) =
    find_bools_bool b
 @ find_bools_list l
and find_bools_int _ = []
and find_bools_bool b = [b]
and find_bools_list [] =
  [] → []
| i :: is → find_bools_int i
          @ find_bools_list is
```

Extensions: trimming branches

Collecting bools

```
everything (bool * list int)
  (@) (mkQ Typeable.bool [] (fun x → [x]))
```

```
let find_bools_pair (b, l) = [b]
```

Next time: unikernels