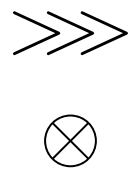
Last time: monads (etc.)

$$\gg=$$

This time: arrows, applicatives (etc.)



Recap: monads, bind and let!

An imperative program

```
let id = !counter in
let () = counter := id + 1 in
  string_of_int id
```

A monadic program

```
get           >>= fun id →
put (id + 1)  >>= fun () →
  return (string_of_int id)
```

Recap: Type parameters and instantiation

monads	parameterised monads
`type 'a t`	`type ('p, 'q, 'a)t`
`let .. in`	$\{P\}\ C\ \{Q\}$

Recap: Higher-order effects with monads

```
val composeM :
 ('a → 'b t) → ('b → 'c t) → ('a → 'c t)
```

```
let composeM f g x =
    f x >>= fun y →
    g y
```

```
val uncurryM :
 ('a → ('b → 'c t) t) → (('a * 'b) → 'c t)
```

```
let uncurryM f (x,y) =
    f x >>= fun g →
    g y
```

Applicatives

(`let x = e ... and`)

Allowing only "static" effects

Idea: stop information flowing from one computation into another.

Only allow unparameterised computations:

$$1 \rightsquigarrow b$$

We can no longer write functions like this:

$$\mathsf{composeE} \quad : \quad (a \rightsquigarrow b) \to (b \rightsquigarrow c) \to (a \rightsquigarrow c)$$

but some useful functions are still possible:

$$\mathtt{pairE}_{\mathtt{static}} \ : \ (1 \rightsquigarrow a) \to (1 \rightsquigarrow b) \to (1 \rightsquigarrow a \times b)$$

Applicative programs

An imperative program

```
let x = fresh_name ()
and y = fresh_name ()
 in (x, y)
```

An applicative program

```
    pure (fun x y → (x, y))
⊗ fresh_name
⊗ fresh_name
```

Applicatives

```
module type APPLICATIVE =
sig
 type 'a t
 val pure : 'a → 'a t
 val (⊗) : ('a → 'b) t → 'a t → 'b t
end
```

Applicatives

```
module type APPLICATIVE =
sig
 type 'a t
 val pure : 'a → 'a t
 val (⊗) : ('a → 'b) t → 'a t → 'b t
end
```

Laws:

$$\begin{aligned}
\text{pure f} \otimes \text{pure v} &\equiv \text{pure (f v)} \\
\text{u} &\equiv \text{pure id} \otimes \text{u} \\
\text{u} \otimes (\text{v} \otimes \text{w}) &\equiv \text{pure compose} \otimes \text{u} \otimes \text{v} \otimes \text{w} \\
\text{v} \otimes \text{pure x} &\equiv \text{pure (fun f} \rightarrow \text{f x)} \otimes \text{v}
\end{aligned}$$

$\gg=$ vs $\otimes$

The type of $\gg=$: `'a t → ('a → 'b t) → 'b t`

`'a → 'b t`: a function that builds a computation

(Almost) the type of $\otimes$: `'a t → ('a → 'b) t → 'b t`

`('a → 'b) t`: a computation that builds a function

The actual type of $\otimes$: `('a → 'b) t → 'a t → 'b t`

Applicative normal forms

$$\texttt{pure f} \otimes \texttt{c}_1 \otimes \texttt{c}_2 \ \ldots \otimes \texttt{c}_n$$

$$\texttt{pure (fun x}_1\ \texttt{x}_2\ \ldots\ \texttt{x}_n \rightarrow \texttt{e)} \otimes \texttt{c}_1 \otimes \texttt{c}_2 \ \ldots \otimes \texttt{c}_n$$

```
let x1 = c1
and x2 = c2
 ...
and xn = cn
  in e
```

Applicative normalisation via the laws

pure f $\otimes$ (pure g $\otimes$ `fresh_name`) $\otimes$ `fresh_name`

Applicative normalisation via the laws

$$
\begin{aligned}
& \text{pure f} \otimes (\text{pure g} \otimes \texttt{fresh\_name}) \otimes \texttt{fresh\_name} \\
\equiv \quad & \text{(composition law)} \\
& (\text{pure compose} \otimes \text{pure f} \otimes \text{pure g} \otimes \texttt{fresh\_name}) \otimes \texttt{fresh\_name}
\end{aligned}
$$

Applicative normalisation via the laws

$$
\begin{aligned}
& \texttt{pure f} \otimes (\texttt{pure g} \otimes \texttt{fresh\_name}) \otimes \texttt{fresh\_name} \\
\equiv \quad & \text{(composition law)} \\
& (\texttt{pure compose} \otimes \texttt{pure f} \otimes \texttt{pure g} \otimes \texttt{fresh\_name}) \otimes \texttt{fresh\_name} \\
\equiv \quad & \text{(homomorphism law } (\times 2)) \\
& \texttt{pure (compose f g)} \otimes \texttt{fresh\_name} \otimes \texttt{fresh\_name}
\end{aligned}
$$

Creating applicatives: every monad is an applicative

```
module Applicative_of_monad (M:MONAD) :
  APPLICATIVE with type 'a t = 'a M.t =
struct
  type 'a t = 'a M.t
  let pure = M.return
  let (⊗) f p =
     M.(f >>= fun g →
        p >>= fun q →
        return (g q))
end
```

The state applicative via the state monad

```
module StateA(S : sig type t end) :
sig
  type state = S.t
  include APPLICATIVE
  val get : state t
  val put : state → unit t
  val runState : 'a t → init:state → state * 'a
end =
struct
  type state = S.t
  include Applicative_of_monad(State(S))
  let (get, put, runState) = M.(get, put, runState)
end
```

Creating applicatives: composing applicatives

```
module Compose (F : APPLICATIVE)
               (G : APPLICATIVE) :
  APPLICATIVE with type 'a t = 'a G.t F.t =
struct
  type 'a t = 'a G.t F.t
  let pure x = F.pure (G.pure x)
  let (⊗) f x = F.(pure G.(⊗) ⊗ f ⊗ x)
end
```

Creating applicatives: the dual applicative

```
module Dual_applicative (A: APPLICATIVE)
  : APPLICATIVE with type 'a t = 'a A.t =
struct
  type 'a t = 'a A.t
  let pure = A.pure
  let (⊗) f x =
    A.(pure (fun y g → g y) ⊗ x ⊗ f)
end


module RevNameA = Dual_applicative(NameA)

RevNameA.(pure (fun x y → (x, y))
        ⊗ fresh_name
        ⊗ fresh_name)
```

Composed applicatives are law-abiding

pure f $\otimes$ pure x

Composed applicatives are law-abiding

$$
\begin{aligned}
& \text{pure f} \otimes \text{pure x} \\
\equiv \quad & \text{(definition of } \otimes \text{ and pure)} \\
& \text{F.pure } (\otimes_G) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)}
\end{aligned}
$$

Composed applicatives are law-abiding

$$
\begin{aligned}
& \text{pure f} \otimes \text{pure x} \\
\equiv \quad & \text{(definition of } \otimes \text{ and pure)} \\
& \text{F.pure } (\otimes_G) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)} \\
\equiv \quad & \text{(homomorphism law for F } (\times 2)) \\
& \text{F.pure (G.pure f} \otimes_G \text{ G.pure x)}
\end{aligned}
$$

Composed applicatives are law-abiding

$$
\begin{aligned}
& \text{pure f} \otimes \text{pure x} \\
\equiv \quad & \text{(definition of } \otimes \text{ and pure)} \\
& \text{F.pure } (\otimes_G) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)} \\
\equiv \quad & \text{(homomorphism law for F } (\times 2)) \\
& \text{F.pure (G.pure f} \otimes_G \text{ G.pure x)} \\
\equiv \quad & \text{(homomorphism law for G)} \\
& \text{F.pure (G.pure (f x))}
\end{aligned}
$$

Composed applicatives are law-abing

$$
\begin{aligned}
& \text{pure f} \otimes \text{pure x} \\
\equiv \quad & \text{(definition of } \otimes \text{ and pure)} \\
& \text{F.pure } (\otimes_G) \otimes_F \text{F.pure (G.pure f)} \otimes_F \text{F.pure (G.pure x)} \\
\equiv \quad & \text{(homomorphism law for F } (\times 2)) \\
& \text{F.pure (G.pure f} \otimes_G \text{ G.pure x)} \\
\equiv \quad & \text{(homomorphism law for G)} \\
& \text{F.pure (G.pure (f x))} \\
\equiv \quad & \text{(definition of pure)} \\
& \text{pure (f x)}
\end{aligned}
$$

Fresh names, monadically

```
type 'a tree =
    Empty : 'a tree
  | Tree : 'a tree * 'a * 'a tree → 'a tree

module IState = State (struct type t = int end)

let fresh_name : string IState.t =
  get          >>= fun i →
  put (i + 1) >>= fun () →
  return (Printf.sprintf "x%d" i)

let rec label_tree : 'a tree → string tree IState.t =
  function
    Empty → return Empty
  | Tree (l, v, r) →
    label_tree l >>= fun l →
    fresh_name   >>= fun name →
    label_tree r >>= fun r →
    return (Tree (l, name, r))
```

Naming as a primitive effect

Problem: we cannot write `fresh_name` using the `APPLICATIVE` interface.

```
let fresh_name : string IState.t =
  get          >>= fun i →
  put (i + 1) >>= fun () →
  return (Printf.sprintf "x%d" i)
```

Solution: introduce it as a primitive effect:

```
module NameA :
sig
  include APPLICATIVE
  val fresh_name : string t
end = ...
```

Traversing with namer

```
let rec label_tree : 'a tree → string tree NameA.t =
  function
    Empty → pure Empty
  | Tree (l, v, r) →
    pure (fun l name r → Tree (l, name, r))
      ⊗ label_tree l
      ⊗ fresh_name
      ⊗ label_tree r
```

The phantom monoid applicative

```
module type MONOID =
sig
  type t
  val zero : t
  val (++) : t → t → t
end

module Phantom_monoid (M: MONOID)
  : APPLICATIVE with type 'a t = M.t =
struct
  type 'a t = M.t
  let pure _ = M.zero
  let (⊗) = M.(++)
end
```

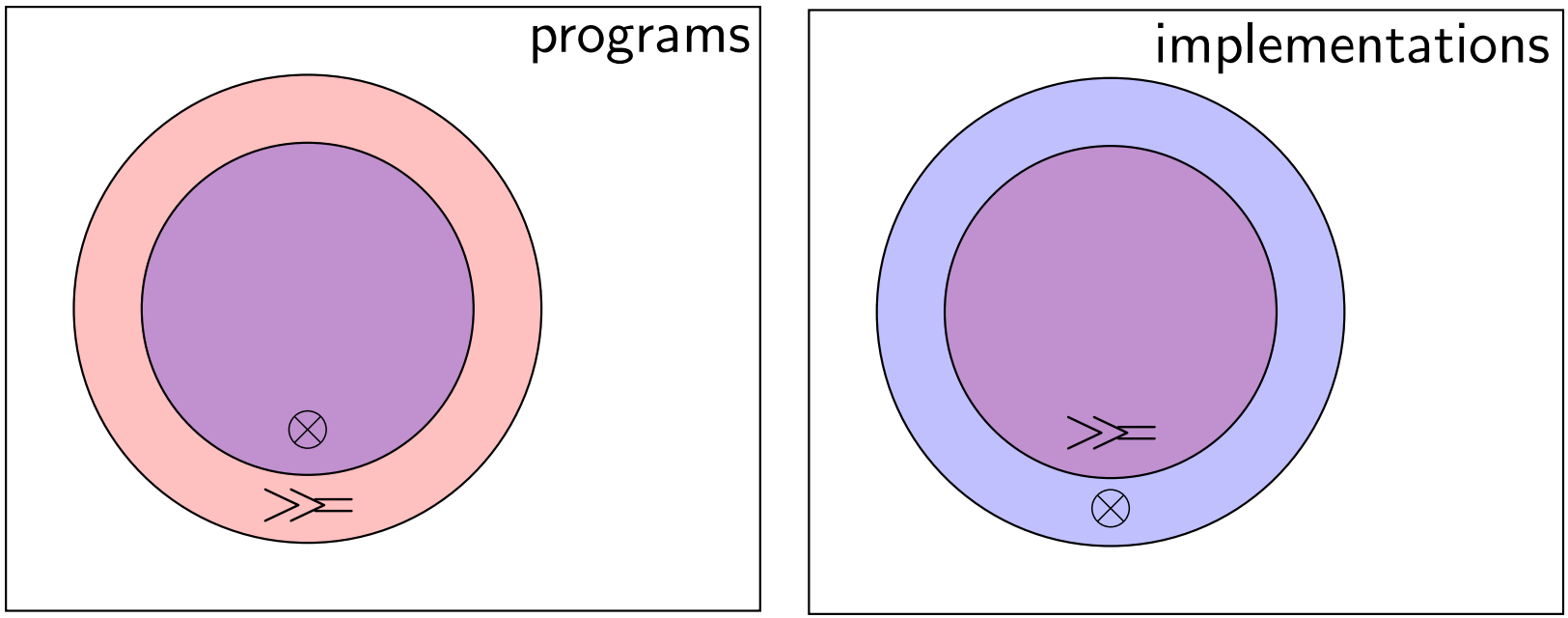
The phantom monoid applicative

```
module type MONOID =
sig
  type t
  val zero : t
  val (++) : t → t → t
end

module Phantom_monoid (M: MONOID)
  : APPLICATIVE with type 'a t = M.t =
struct
  type 'a t = M.t
  let pure _ = M.zero
  let (⊗) = M.(++)
end
```

Observation: we cannot implement `Phantom_monoid` as a monad.

Applicatives vs monads



Some monadic programs are not applicative, e.g. `fresh_name`.

Some applicative instances are not monadic, e.g. `Phantom_monoid`.

Guideline: Postel's law

Be conservative in what you do,
be liberal in what you accept from others.

Guideline: Postel's law

Be conservative in what you do,
be liberal in what you accept from others.

Conservative in what you do: **use** applicatives, not monads.
(Applicatives give the implementor more freedom.)

Guideline: Postel's law

Be conservative in what you do,
be liberal in what you accept from others.

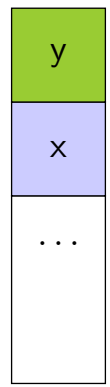
Conservative in what you do: **use** applicatives, not monads.
(Applicatives give the implementor more freedom.)

Liberal in what you accept: **implement** monads, not applicatives.
(Monads give the user more power.)

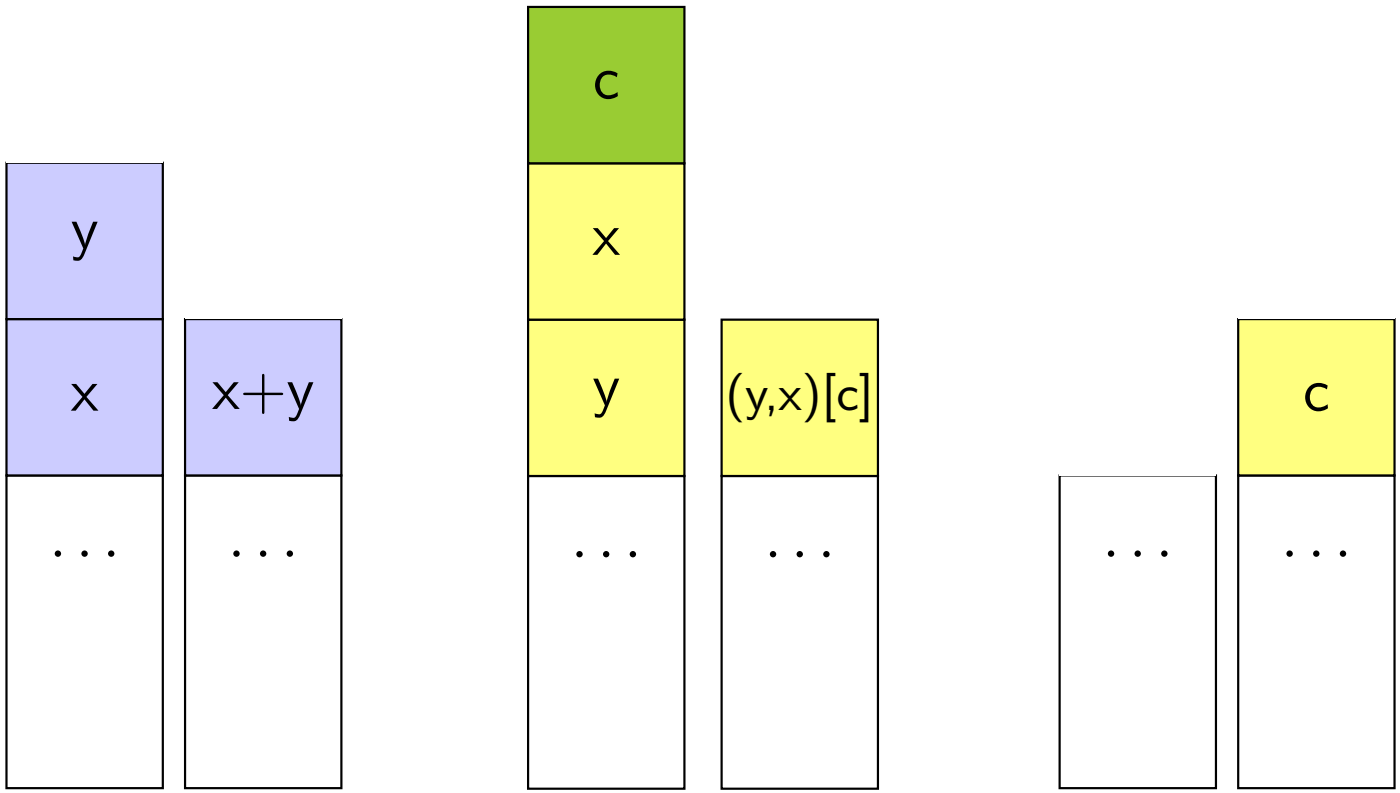
Parameterised applicatives

```
module type PARAMETERISED_APPLICATIVE =
sig
  type ('s,'t,'a) t
  val unit : 'a → ('s,'s,'a) t
  val (⊗) : ('r,'s,'a → 'b) t
          → ('s,'t,'a) t
          → ('r,'t,'b) t
end
```

Stack machines



Recap: stack machine instructions



Stack machine operations

```
module type STACK_OPS =
sig
  type ('s,'t,'a) t
  val add : (int * (int * 's),
                    int * 's, unit) t
  val _if_ : (bool * ('a * ('a * 's)),
                              'a * 's, unit) t
  val push_const : 'a → ('s,
                     'a * 's, unit) t
end
```

Stack machines, monadically

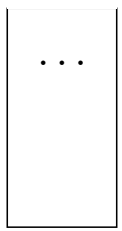
```
module type STACKM = sig
 include PARAMETERISED_MONAD
 include STACK_OPS
   with type ('s,'t,'a) t := ('s,'t,'a) t
 val execute : ('s,'t,'a) t → 's → 't * 'a
end

module StackM : STACKM = struct
 include PState

 let add = get >>= fun (x,(y,s)) → put (x+y,s)
 let _if_ = get (c,(t,(e,s))) >>=
    put (if c then t else e)
 let push_const k = get >>= fun s → put (k, s)
 let execute = runState
end
```

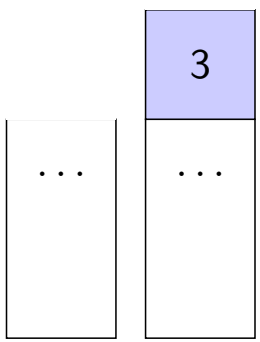
Programming the monadic stack machine

```
push_const 3     >>= fun () →
push_const 4     >>= fun () →
push_const 5     >>= fun () →
push_const true  >>= fun () →
_if_             >>= fun () →
add              >>= fun () →
return ()
```



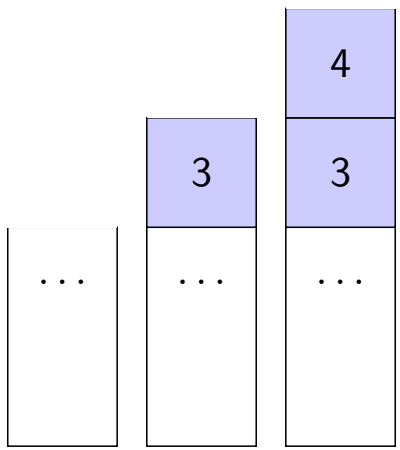
Programming the monadic stack machine

```
push_const 3     >>= fun () →
push_const 4     >>= fun () →
push_const 5     >>= fun () →
push_const true  >>= fun () →
_if_             >>= fun () →
add              >>= fun () →
return ()
```



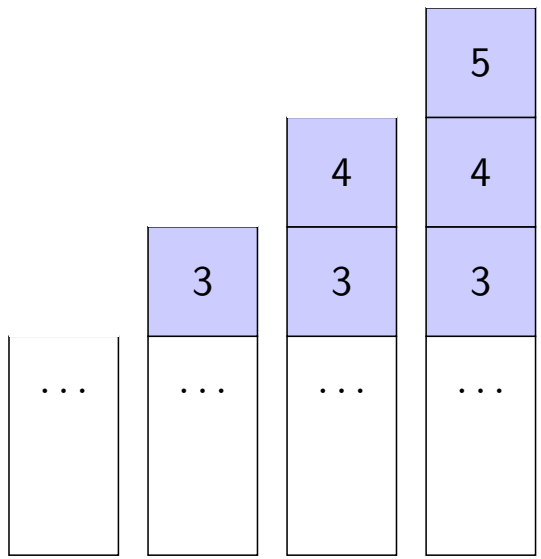
Programming the monadic stack machine

```
push_const 3     >>= fun () →
push_const 4     >>= fun () →
push_const 5     >>= fun () →
push_const true  >>= fun () →
_if_             >>= fun () →
add              >>= fun () →
return ()
```



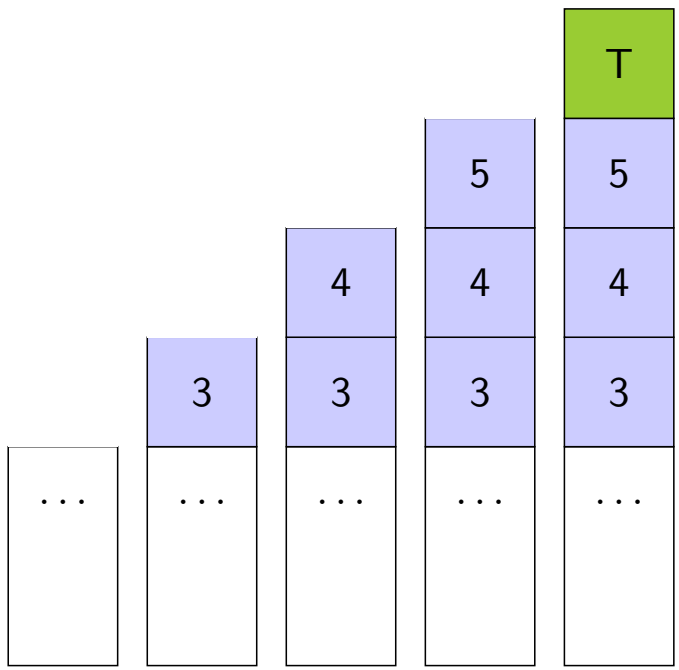
Programming the monadic stack machine

```
push_const 3     >>= fun () →
push_const 4     >>= fun () →
push_const 5     >>= fun () →
push_const true  >>= fun () →
_if_             >>= fun () →
add              >>= fun () →
return ()
```



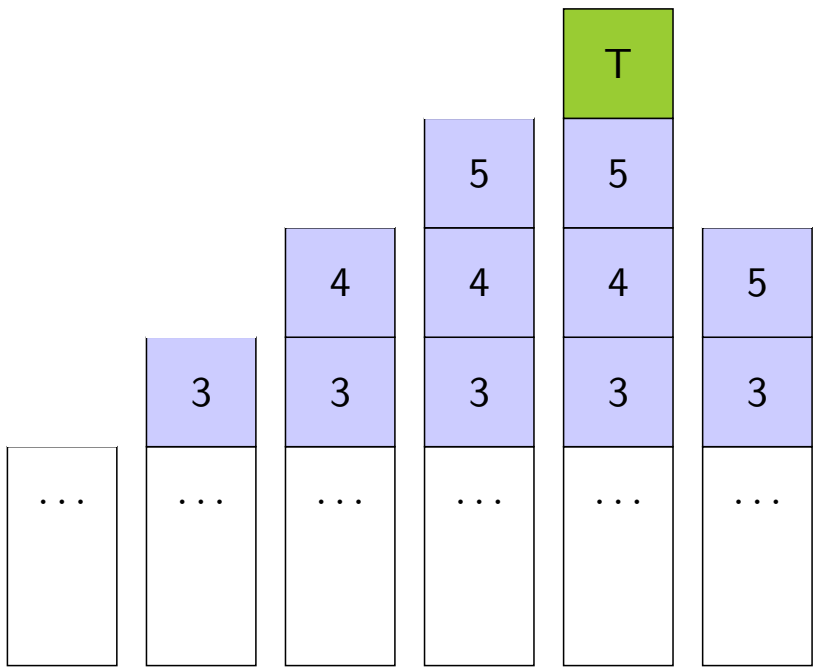
Programming the monadic stack machine

```
push_const 3     >>= fun () →
push_const 4     >>= fun () →
push_const 5     >>= fun () →
push_const true  >>= fun () →
_if_             >>= fun () →
add              >>= fun () →
return ()
```



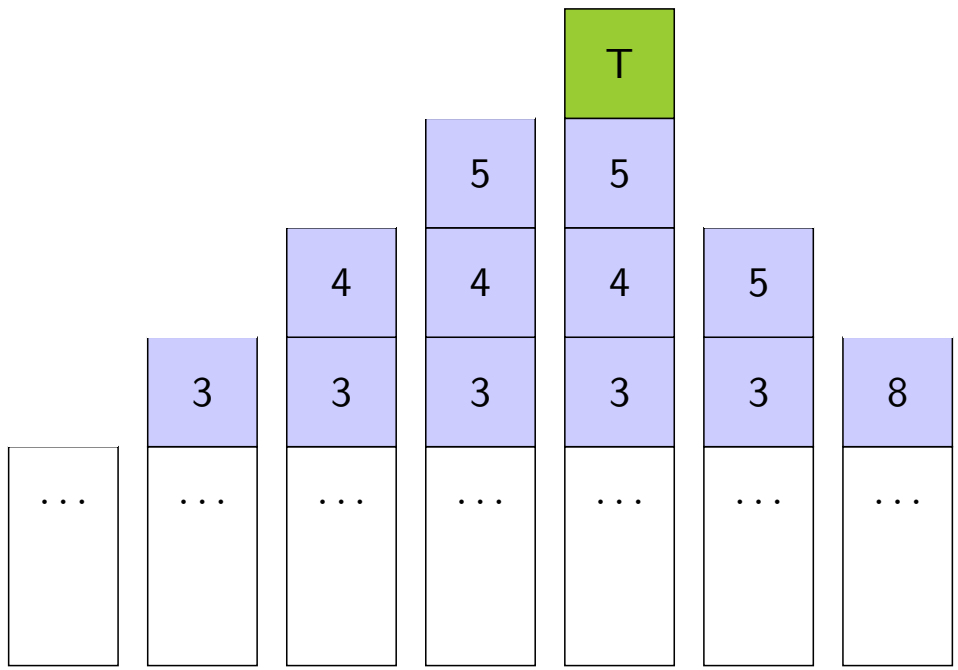
Programming the monadic stack machine

```
push_const 3     >>= fun () →
push_const 4     >>= fun () →
push_const 5     >>= fun () →
push_const true  >>= fun () →
_if_             >>= fun () →
add              >>= fun () →
return ()
```



Programming the monadic stack machine

```
push_const 3     >>= fun () →
push_const 4     >>= fun () →
push_const 5     >>= fun () →
push_const true  >>= fun () →
_if_             >>= fun () →
add              >>= fun () →
return ()
```



Stack machines, applicatively

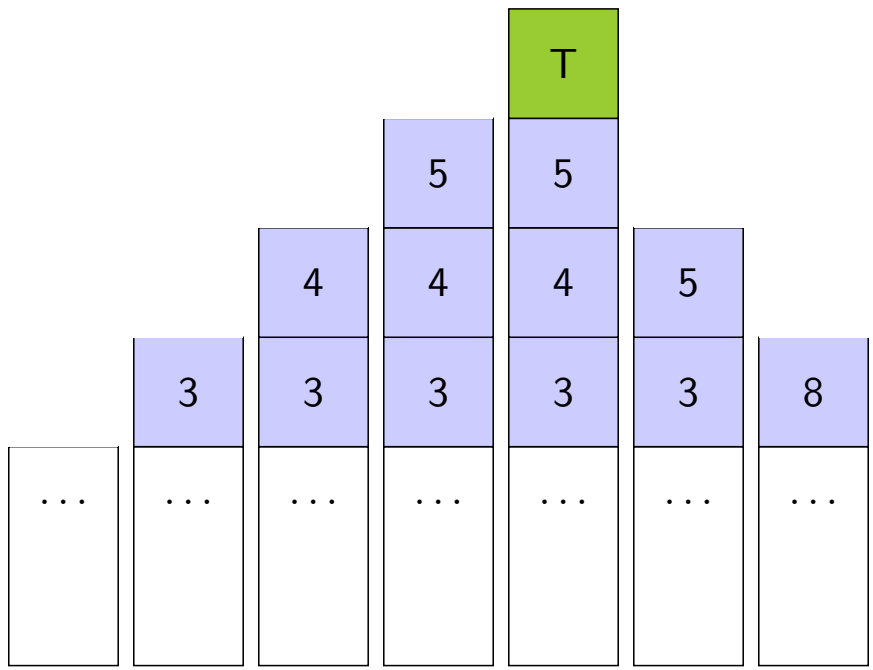
```
module type STACKA = sig
 include PARAMETERISED_APPLICATIVE
 include STACK_OPS
   with type ('s,'t,'a) t := ('s,'t,'a) t
 val execute : ('s,'t,'a) t → 's → 't
end

module StackA : STACKA = struct
 include Applicative_of_monad(StackM)

 let (add, _if_, push_const) =
   StackM.(add, _if_, push_const)
 let execute m s = fst (StackM.execute m s)
end
```

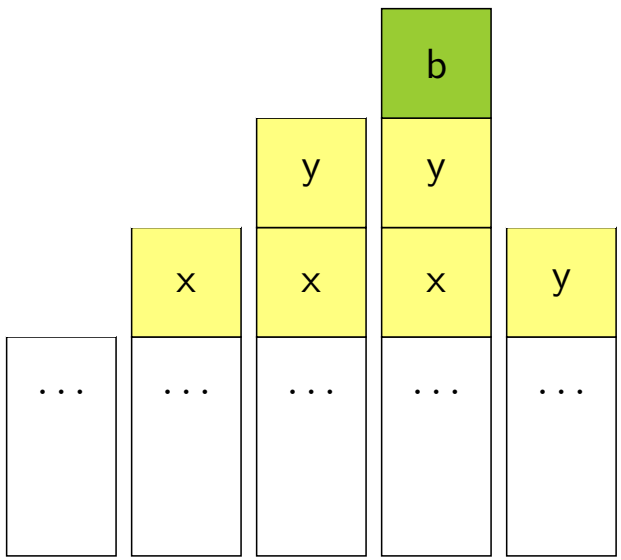
Programming the applicative stack machine

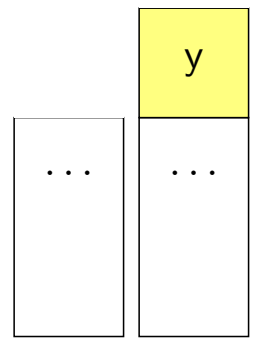
```
pure (fun () () () () () () → ())
⊗ push_const 3
⊗ push_const 4
⊗ push_const 5
⊗ push_const true
⊗ _if_
⊗ add
```



Optimising stack machines

```
PushConst x :: PushConst y :: PushConst true :: If ⇝ PushConst y
```





First-order stack machines, applicatively

```
let rec (++) :
 type r s t.(r,s) instrs → (s,t) instrs → (r,t) instrs =
 fun l r → match l with
   Stop → r
 | i :: is → i :: is ++ r

module StackA1 : STACKA = struct
  type ('s, 't, 'a) t = ('s, 't) instrs
  let pure a = Stop
  let (⊗) = (++)
  let add = Add :: Stop
  let _if_ = If :: Stop
  let push_const v = PushConst v :: Stop
  let execute = (* ... *)
end
```

Optimising stack machines

```
let rec opt : type s t.(s,t) instrs → (s,t) instrs =
 function
   [] →
      []
| PushConst x :: PushConst y :: PushConst c ::
   If :: s →
     opt (PushConst (if c then y else x) :: s)
| i :: is →
     i :: opt is
```

First-order stack machines, applicatively

```
module StackA1 : STACKA = struct
  type ('s, 't, 'a) t = ('s, 't) instrs
  let pure a = Stop
  let (⊗) l r = opt (l ++ r)
  let add = Add :: Stop
  let _if_ = If :: Stop
  let push_const v = PushConst v :: Stop
  let execute = (* ... *)
end
```

Monoids

$(;)$

Instantiating applicatives

```
module type MONOID =
sig
  type t
  val zero : t
  val (++) : t → t → t
end
```

M_1 ;
M_2 ;
... ;
M_n

Instantiating applicatives

```
module type MONOID =
sig
  type t
  val zero : t
  val (++) : t → t → t
end
```

M_1 ;
M_2 ;
... ;
M_n

Laws:

$$\begin{aligned} \mathsf{zero} \mathbin{+\!\!+} \mathsf{m} &\equiv \mathsf{m} \\ \mathsf{m} \mathbin{+\!\!+} \mathsf{zero} &\equiv \mathsf{m} \\ (\mathsf{m} \mathbin{+\!\!+} \mathsf{n}) \mathbin{+\!\!+} \mathsf{o} &\equiv \mathsf{m} \mathbin{+\!\!+} (\mathsf{n} \mathbin{+\!\!+} \mathsf{o}) \end{aligned}$$

Arrows

(`let x = c e ... in`)

Arrows for first-order computation

```
let x1 = C1 e1 in
let x2 = C2 e2 in
    ...
let xn = Cn en in
    e
```

$$\frac{\Gamma; \Delta \vdash M : A \qquad \Gamma; \Delta, x : A \vdash N : B}{\Gamma; \Delta \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : B}$$

$$\frac{\Gamma \vdash M : A \rightsquigarrow B \qquad \Gamma, \Delta \vdash N : A}{\Gamma; \Delta \vdash M\ N : B}$$

Programming with arrows

An imperative program

```
let id = !counter in
let () = counter := id + 1 in
  string_of_int id
```

A program with arrows

```
get () >>>
arr (fun id → (id+1, id)) >>>
first put >>>
arr (fun ((), id) → string_of_int id)
```

Arrows

```
module type ARROW =
sig
 type ('a, 'b) t
 val arr : ('a → 'b) → ('a, 'b) t
 val (⋙) : ('a, 'b) t → ('b, 'c) t → ('a, 'c) t
 val first : ('a, 'b) t → ('a * 'c, 'b * 'c) t
end
```

Arrows

```
module type ARROW =
sig
 type ('a, 'b) t
 val arr : ('a → 'b) → ('a, 'b) t
 val (>>>) : ('a, 'b) t → ('b, 'c) t → ('a, 'c) t
 val first : ('a, 'b) t → ('a * 'c, 'b * 'c) t
end
```

Laws:

$$\begin{aligned} \text{arr f} \ggg \text{arr g} &\equiv \text{arr (compose g f)} \\ (\text{f} \ggg \text{g}) \ggg h &\equiv \text{f} \ggg (\text{g} \ggg h) \\ \text{arr id} \ggg \text{f} &\equiv \text{f} \\ \ldots &\quad \ldots \end{aligned}$$

$\gg=$ vs $\otimes$ vs $\ggg$

The type of $\gg=$: `'a t → ('a → 'b t) → 'b t`

`'a → 'b t`: a function that builds a computation

The type of $\otimes$: `('a → 'b) t → 'a t → 'b t`

`('a → 'b) t`: a computation that builds a function

The type of $\ggg$: `('a, 'b) t → ('b, 'c) t → ('a, 'c) t`

`('a, 'b) t`: a computation with both input and output

Creating arrows: every monad yields an arrow

```
module Arrow_of_monad (M: MONAD) :
  ARROW with type ('a, 'b) t = 'a -> 'b M.t =
struct
  type ('a, 'b) t = 'a -> 'b M.t
  let arr f x = M.return (f x)
  let (>>>) f g x = M.(f x >>= fun y -> g y)
  let first f (x,y) =
    M.(f x >>= fun z -> return (z, y))
end
```

Arrows and fresh_name

```
module State_arrow (S: sig type t end) :
sig
  include ARROW
  val get : (unit, S.t) t
  val put : (S.t, unit) t
end =
struct
  module M = State(S)
  include Arrow_of_monad(M)
  let get, put = M.((fun () -> get), put)
end


module IState_arrow =
  State_arrow(struct type t = int end)

let fresh_name : (unit, string) State_arrow.t =
  get >>> arr (fun s → (s+1, s)) >>>
  first put >>> arr (fun ((), s) → sprintf "x%d" s)
```

Arrows and fresh_name, continued

```
val fresh_name : (unit, string) IState_arrow.t
```

```
let rec label_tree :
  'a. 'a tree -> (unit, string tree) IState_arrow.t =
  function
    Empty -> arr (fun () -> Empty)
  | Tree (l, v, r) ->
    label_tree l >>>
    arr (fun l -> ((), l)) >>>
    first fresh_name >>>
    arr (fun (n, l) -> ((), (n, l))) >>>
    first (label_tree r) >>>
    arr (fun (r, (n, l)) -> Tree (l, n, r))
```

Arrows and uncurryM

uncurryM with monads

```
val uncurryM :
 ('a → ('b → 'c t) t) → (('a * 'b) → 'c t)

let uncurryM f (x,y) =
    f x >>= fun g →
    g y >>= fun z →
    return z
```

uncurryM with let

```
let uncurryM f (x, y) =
  let g = f x in
  let z = g y in
    z
```

uncurryM with arrows ...

Arrows and uncurryM

`uncurryM` with monads

```
val uncurryM :
 ('a → ('b → 'c t) t) → (('a * 'b) → 'c t)

let uncurryM f (x,y) =
    f x >>= fun g →
    g y >>= fun z →
    return z
```

`uncurryM` with `let`

```
let uncurryM f (x, y) =
  let g = f x in
  let z = g y in
    z
```

`uncurryM` with arrows . . .

. . . is impossible, because there is a *control dependency*

Using arrows: every arrow yields an applicative

```
module Applicative_of_arrow (A: ARROW) :
  APPLICATIVE with type 'a t = (unit, 'a) A.t =
struct
  type 'a t = (unit, 'a) A.t
  let pure x = A.arr (fun () -> x)
  let (<*>) f p =
    A.(f >>> arr (fun g -> ((), g)) >>>
       first p >>> arr (fun (y, g) -> (g y)))
end
```

Arrows and monads: not every arrow is a monad

```
module Two_phase(M: MONAD) (N: MONAD) :
  ARROW with type ('a, 'b) t = ('a -> 'b N.t) M.t =
struct
  type ('a, 'b) t = ('a -> 'b N.t) M.t
  let arr f = M.return (fun x -> N.return (f x))
  let (>>>) f g =
  M.(f >>= fun h ->
     g >>= fun k ->
     return N.(fun a -> h a >>= k))
  let first f =
    M.(f >>= fun h ->
       return N.(fun (a, c) ->
                   h a >>= fun b -> return (b, c)))
end
```

Arrows and applicatives again

Reversing effect order with applicatives

```
module Dual_applicative (A: APPLICATIVE)
  : APPLICATIVE with type 'a t = 'a A.t =
struct
  type 'a t = 'a A.t
  let pure = A.pure
  let (⊗) f x =
    A.(pure (fun y g → g y) ⊗ x ⊗ f)
end
```

Reversing effect order with monads . . .

Reversing effect order with arrows. . .

Arrows and applicatives again

Reversing effect order with applicatives

```
module Dual_applicative (A: APPLICATIVE)
  : APPLICATIVE with type 'a t = 'a A.t =
struct
  type 'a t = 'a A.t
  let pure = A.pure
  let (⊗) f x =
    A.(pure (fun y g → g y) ⊗ x ⊗ f)
end
```

Reversing effect order with monads . . .

. . . is impossible, because computations have *control dependencies*

Reversing effect order with arrows. . .

Arrows and applicatives again

Reversing effect order with applicatives

```
module Dual_applicative (A: APPLICATIVE)
  : APPLICATIVE with type 'a t = 'a A.t =
struct
  type 'a t = 'a A.t
  let pure = A.pure
  let (⊗) f x =
    A.(pure (fun y g → g y) ⊗ x ⊗ f)
end
```

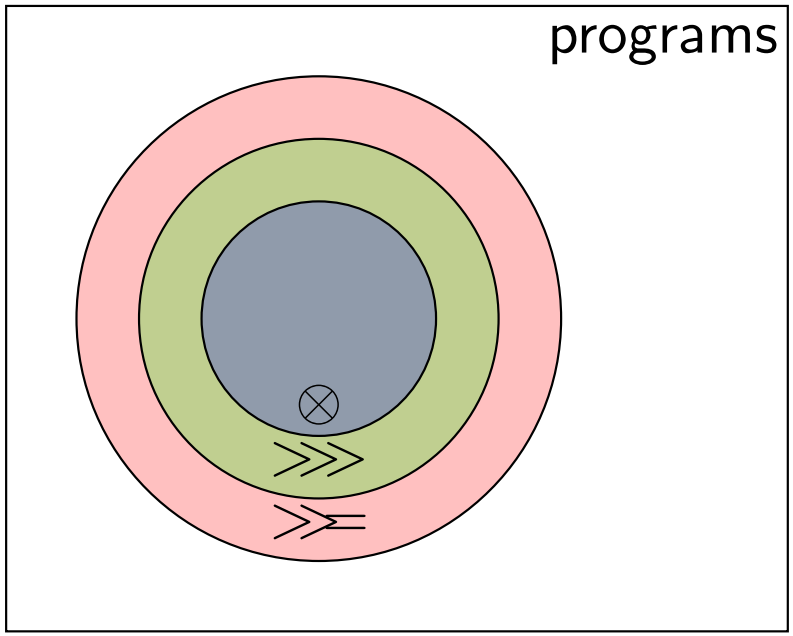
Reversing effect order with monads . . .

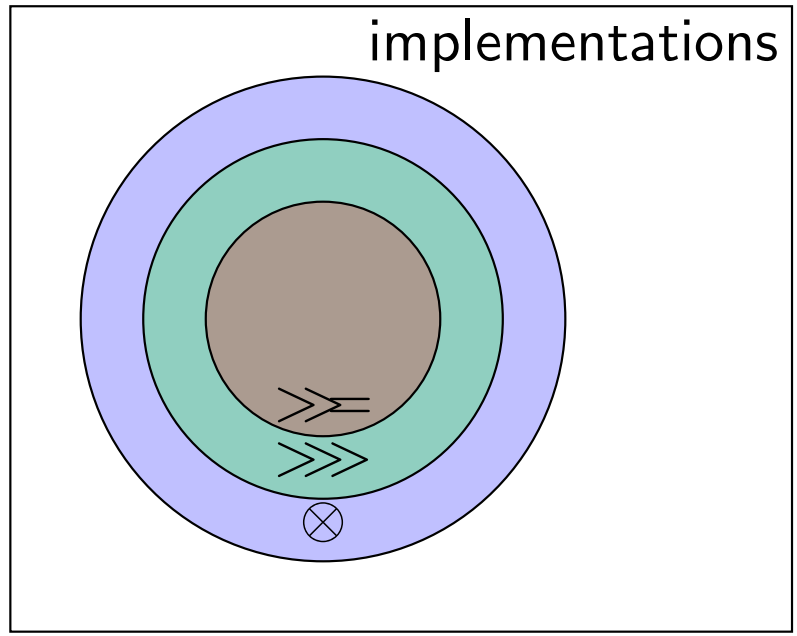
. . . is impossible, because computations have *control dependencies*

Reversing effect order with arrows. . .

. . . is impossible, because computations have *data dependencies*

Applicatives vs arrows vs monads





Some monadic programs can't be written with arrows

- e.g. `uncurryM`

Some arrow programs can't be written with applicatives

- e.g. `fresh_name`

Some applicative instances can't be written as arrows

- e.g. `Dual_applicative`

Some arrow instances can't be written as monads

- e.g. `Staged_arrow`.

Summary

monads

```
let x1 = e1 in
let x2 = e2 in
   ...
let xn = en in
   e
```

arrows

```
let x1 = C1 e1 in
let x2 = C2 e2 in
   ...
let xn = Cn en in
   e
```

applicatives

```
let x1 = e1
and x2 = e2
   ...
and xn = en in
   e
```

monoids

```
e1 ;
e2 ;
... ;
en
```

parameterised monads and applicatives

$$\{P\}\ C\ \{Q\}$$

Next time: generic programming

```
val show : 'a → string
```