Last time: monads (etc.)

$\gg=$

This time: applicatives (etc.)

$$\otimes$$

Example effects

Effects available in OCaml

(higher-order) state

r := f; !r ()

exceptions

raise Not_found

I/O of various sorts

input_byte stdin

concurrency (interleaving)

Gc. finalise v f

non-termination

let rec f x = f x

Effects unavailable in OCaml

non-determinism

amb f g h

first-class continuations

escape x in e

polymorphic state

r := "one"; r := 2

checked exceptions

$\text{int} \xrightarrow{\text{IOError}} \text{bool}$

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)
```

Type parameters and instantiation

monads	indexed monads	parameterised monads
`type 'a t`	`type ('e, 'a) t`	`type ('s, 't, 'a) t`
`let .. in`	$\Gamma \vdash M : A \; ! \; e$	{P} C {Q}

Monadic effect are higher-order

$$
\begin{aligned}
\text{composeE} &: (a \xrightarrow{E} b) \to (b \xrightarrow{E} c) \to (a \xrightarrow{E} c) \\
\text{pairE} &: (a \xrightarrow{E} b) \to (c \xrightarrow{E} d) \to (a \times c \xrightarrow{E} b \times d) \\
\text{uncurryE} &: (a \xrightarrow{E} b \xrightarrow{E} c) \to (a \times b \xrightarrow{E} c) \\
\text{liftPure} &: (a \to b) \to (a \xrightarrow{E} b)
\end{aligned}
$$

Higher-order effects with monads

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

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

Applicatives

(let ... and)

Allowing only "static" effects

Idea: stop information flowing from one computation into another.

Only allow unparameterised computations:

$$1 \xrightarrow{E} b$$

We can no longer write functions like this:

$$\text{composeE} \quad : \quad (a \xrightarrow{E} b) \to (b \xrightarrow{E} c) \to (a \xrightarrow{E} c)$$

but some useful functions are still possible:

$$\text{pairE}_{\text{static}} \quad : \quad (1 \xrightarrow{E} a) \to (1 \xrightarrow{E} b) \to (1 \xrightarrow{E} a \times b)$$

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

$$\text{pure } f \otimes c_1 \otimes c_2 \ldots \otimes c_n$$

$$\text{pure } (\textbf{fun } x_1\ x_2 \ldots x_n \to e) \otimes c_1 \otimes c_2 \ldots \otimes c_n$$

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

Applicative normalisation via the laws

$\text{pure f} \otimes (\text{pure g} \otimes \text{fresh\_name}) \otimes \text{fresh\_name}$

Applicative normalisation via the laws

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

Applicative normalisation via the laws

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

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
```

Composed applicatives are law-abiding

$$\text{pure f} \otimes \text{pure x}$$

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

pure f $\otimes$ pure x

$\equiv$ (definition of $\otimes$ and pure)

F.pure ($\otimes_G$) $\otimes_F$ F.pure (G.pure f) $\otimes_F$ F.pure (G.pure x)

$\equiv$ (homomorphism law for F ($\times 2$))

F.pure (G.pure f $\otimes_G$ G.pure x)

Composed applicatives are law-abiding

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

Composed applicatives are law-abiding

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

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
```

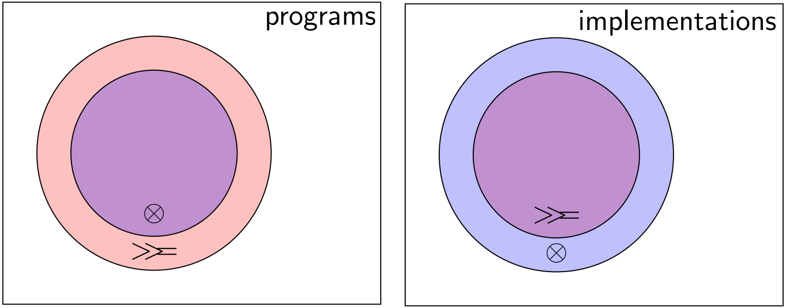
The phantom monoid applicative

```
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.)

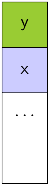
Liberal in what you accept: **implement** monads, not applicatives.
(Monads give the user more power.)

Parameterised and indexed applicatives

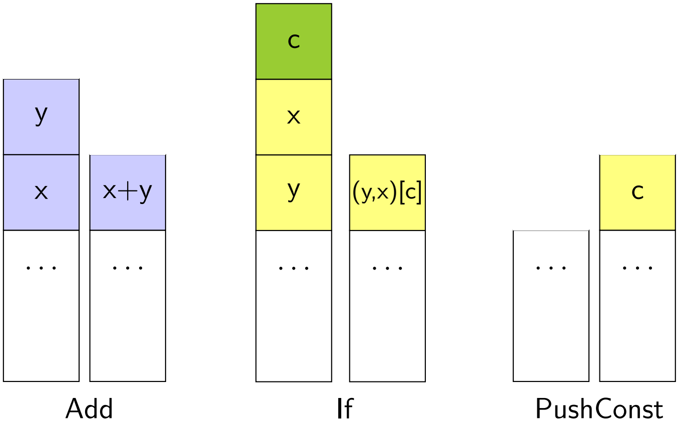
```
module type PARAMETERISED_APPLICATIVE =
sig
  type ('s,'t,'a) t
  val unit : 'a → ('s,'s,'a) t
  val (⊗) : ('r,'s,'a → 'b) t
          → ('s,'t,'a) t
          → ('r,'t,'b) t
end

module type INDEXED_APPLICATIVE =
sig
  type ('e,'a) t
  val pure : 'a → ('e,'a) t
  val (⊗) : ('e,'a → 'b) t
          → ('e,'a) t
          → ('e,'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

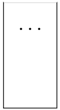
```
module type STACKM = sig
 include PARAMETERISED_MONAD
 include STACK_OPS
   with type ('s,'t,'a) t := ('s,'t,'a) t
 val execute : ('s,'t,'a) t → 's → 't * 'a
end

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
```

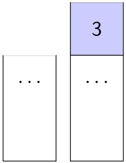
Programming the monadic stack machine

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



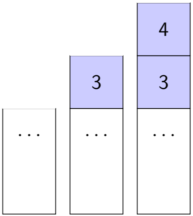
Programming the monadic stack machine

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



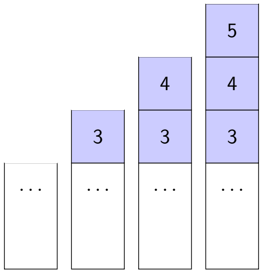
Programming the monadic stack machine

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



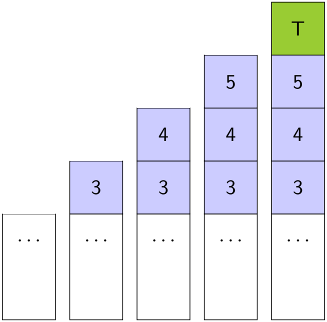
Programming the monadic stack machine

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



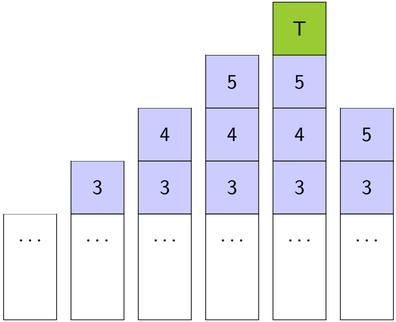
Programming the monadic stack machine

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



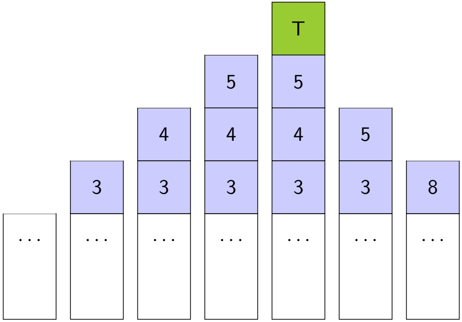
Programming the monadic stack machine

```
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

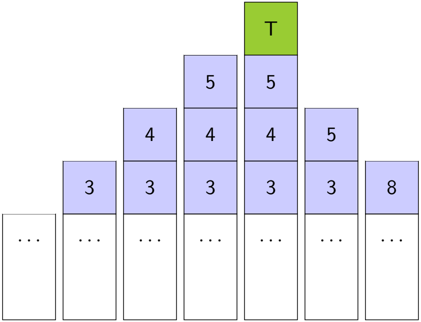
```
module type STACKA = sig
 include PARAMETERISED_APPLICATIVE
 include STACK_OPS
   with type ('s,'t,'a) t := ('s,'t,'a) t
 val execute : ('s,'t,'a) t → 's → 't
end

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
```

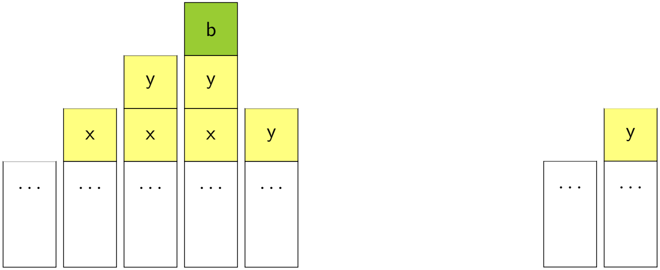
Programming the applicative stack machine

```
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 ;
$\ldots$;
M_n

Summary

monads	applicatives	monoids

`let! x1 = M1 in`
`let! x2 = M2 in`
`...`
`let! xn = Mn in`
`N`
	`let! x1 = M1`
	`and! x2 = M2`
	`...`
	`and! xn = Mn in`
	`N`
		`M1 ;`
		`M2 ;`
		`... ;`
		`Mn`

indexed monads and applicatives

$$\Gamma \vdash M : A \mathbin{!} e$$

parameterised monads and applicatives

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