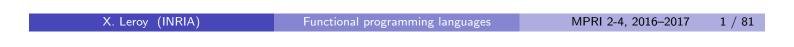
# Functional programming languages Part IV: monadic transformations, monadic programming

Xavier Leroy

INRIA Paris-Rocquencourt

MPRI 2-4, 2016-2017

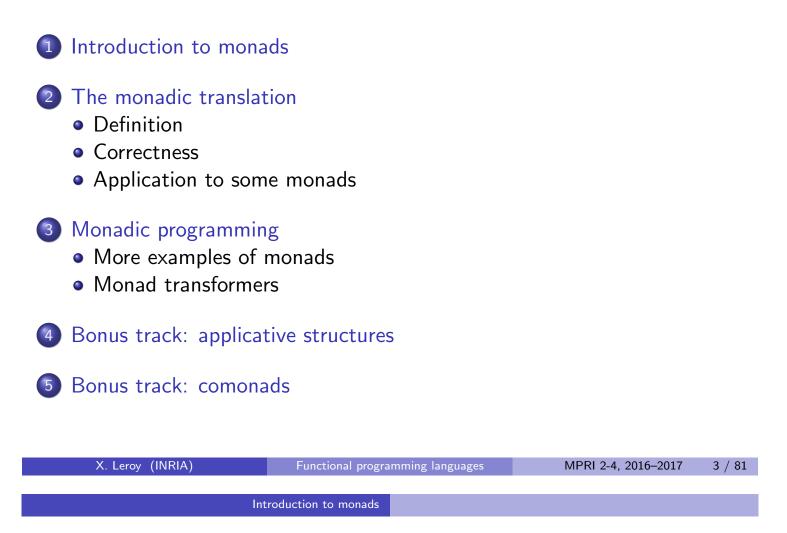


# Monads in programming language theory

Monads are a technical device (inspired from category theory) with several uses in programming:

- To structure denotational semantics and make them easy to extend with new language features. (E. Moggi, 1989.) Not treated in this lecture.
- To factor out commonalities between many program transformations and between their proofs of correctness.
- As a powerful programming techniques in pure functional languages. (P. Wadler, 1992; the Haskell community).

# Outline



## Commonalities between program transformations

Consider the conversions to exception-returning style, state-passing style, and continuation-passing style. For constants, variables and  $\lambda$ -abstractions, we have:

$$\begin{bmatrix} N \end{bmatrix} = V(N) \qquad \begin{bmatrix} N \end{bmatrix} = \lambda s.(N,s) \qquad \begin{bmatrix} N \end{bmatrix} = \lambda k.k N \\ \begin{bmatrix} x \end{bmatrix} = V(x) \qquad \begin{bmatrix} x \end{bmatrix} = \lambda s.(x,s) \qquad \begin{bmatrix} x \end{bmatrix} = \lambda k.k x \\ \begin{bmatrix} \lambda x.a \end{bmatrix} = V(\lambda x.\llbracket a \rrbracket) \qquad \begin{bmatrix} \lambda x.a \rrbracket = \lambda s.(\lambda x.\llbracket a \rrbracket, s) \qquad \llbracket \lambda x.a \rrbracket = \lambda k.k \ (\lambda x.\llbracket a \rrbracket)$$

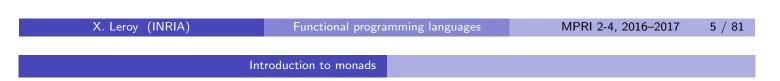
in all three cases, we return (put in some appropriate wrapper) the values N or x or  $\lambda x.[[a]]$ .

#### Commonalities between program transformations

For let bindings, we have:

$$\begin{bmatrix} [\texttt{let } x = a \texttt{ in } b] \end{bmatrix} = \texttt{match } \llbracket a \rrbracket \texttt{ with } E(x) \to E(x) \mid V(x) \to \llbracket b \rrbracket$$
$$\begin{bmatrix} [\texttt{let } x = a \texttt{ in } b] \rrbracket = \lambda s.\texttt{ match } \llbracket a \rrbracket s \texttt{ with } (x, s') \to \llbracket b \rrbracket s'$$
$$\begin{bmatrix} \texttt{let } x = a \texttt{ in } b \rrbracket = \lambda k. \llbracket a \rrbracket (\lambda x. \llbracket b \rrbracket k)$$

In all three cases, we extract (one way or another) the value contained in the computation  $[\![a]\!]$ , bind it to the variable x, and proceed with the computation  $[\![b]\!]$ .



# Commonalities between program transformations

Concerning function applications:

$$\llbracket a b \rrbracket = \lambda k. \llbracket a \rrbracket (\lambda v_a. \llbracket b \rrbracket (\lambda v_b. v_a v_b k))$$

We bind  $\llbracket a \rrbracket$  to a variable  $v_a$ , then bind  $\llbracket b \rrbracket$  to a variable  $v_b$ , then perform the application  $v_a v_b$ .

## Interface of a monad

A monad is defined by a parameterized type  $\alpha$  mon and operations ret, bind and run, with types:

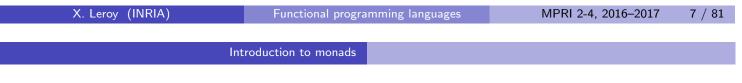
```
\begin{array}{lll} \texttt{ret} & : & \forall \alpha. \; \alpha \to \alpha \; \texttt{mon} \\ \texttt{bind} & : & \forall \alpha, \beta. \; \alpha \; \texttt{mon} \to (\alpha \to \beta \; \texttt{mon}) \to \beta \; \texttt{mon} \\ \texttt{run} & : & \forall \alpha. \; \alpha \; \texttt{mon} \to \alpha \end{array}
```

The type  $\tau$  mon is the type of computations that eventually produce a value of type  $\tau$ .

ret *a* encapsulates a pure expression *a* :  $\tau$  as a trivial computation (of type  $\tau$  mon) that immediately produces the value of *a*.

bind  $a(\lambda x.b)$  performs the computation  $a: \tau \mod$ , binds its value to  $x: \tau$ , then performs the computation  $b: \tau' \mod$ .

run *a* is the execution of a monadic program *a*, extracting its return value.



Monadic laws

The ret and bind operations of the monad are supposed to satisfy the following algebraic laws:

```
bind (ret a) f \approx f a
bind a (\lambda x. ret x) \approx a
bind (\lambda x. b) (\lambda y. c) \approx bind a (\lambda x. bind b (\lambda y. c))
```

The relation  $\approx$  needs to be made more precise, but intuitively means "behaves identically".

# Example: the Exception monad

(also called the Error monad)

```
type \alpha mon = V of \alpha | E of exn
ret a = V(a)
bind m f = match m with E(x) -> E(x) | V(x) -> f x
run m = match m with V(x) -> x
```

bind encapsulates the propagation of exceptions in compound expressions such as  $a \ b$  or let bindings.

Additional operations in this monad:

```
raise x = E(x)
trywith m f = match m with E(x) \rightarrow f x | V(x) \rightarrow V(x)
```

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4, 2016–2017	9 / 81
Int	roduction to monads		

Example: the State monad

```
type \alpha mon = state \rightarrow \alpha \times state
ret a = \lambdas. (a, s)
bind m f = \lambdas. match m s with (x, s') -> f x s'
run m = match m empty_store with (x, s) -> x
```

bind encapsulates the threading of the state in compound expressions.

Additional operations in this monad:

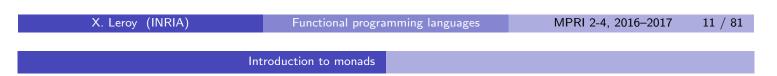
```
ref x = \lambdas. store_alloc x s
deref r = \lambdas. (store_read r s, s)
assign r x = \lambdas. ((), store_write r x s)
```

#### Example: the Continuation monad

type  $\alpha$  mon = ( $\alpha \rightarrow answer$ )  $\rightarrow answer$ ret a =  $\lambda k$ . k a bind m f =  $\lambda k$ . m ( $\lambda v$ . f v k) run m = m ( $\lambda x$ . x)

Additional operations in this monad:

callcc f =  $\lambda k$ . f k k throw x y =  $\lambda k$ . x y



# Alternate presentation of a monad

The alternate presentation replaces bind by two operations fmap and join:

 $\begin{array}{lll} \texttt{ret} & : & \forall \alpha. \; \alpha \to \alpha \; \texttt{mon} \\ \texttt{fmap} & : & \forall \alpha, \beta. \; (\alpha \to \beta) \to (\alpha \; \texttt{mon} \to \beta \; \texttt{mon}) \\ \texttt{join} & : & \forall \alpha, \; (\alpha \; \texttt{mon}) \; \texttt{mon} \to \alpha \; \texttt{mon} \end{array}$ 

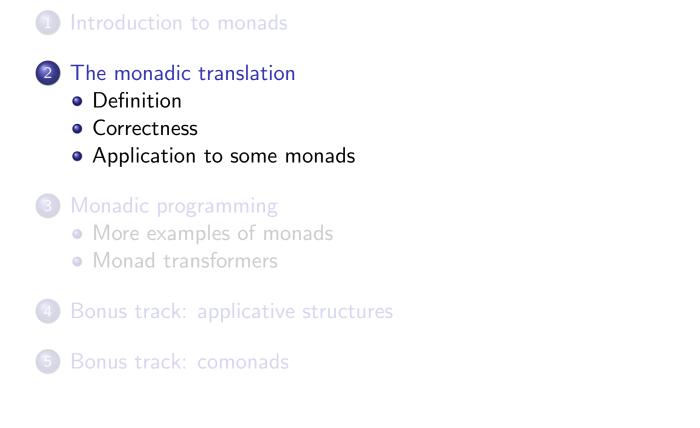
The two presentations are related as follows:

bind 
$$a f \equiv join(fmap f a)$$
  
fmap  $f m \equiv bind m (\lambda x. ret(f x))$   
join  $mm \equiv bind mm (\lambda m. m)$ 

The alternate presentation is closer to category theory but less convenient for programming,

X. Leroy (INRIA)

#### Outline



X. Leroy (INRIA)	Functional programming languages		MPRI 2-4, 2016–2017	13 / 81
Th	e monadic translation	Definition		

## The monadic translation

Core constructs

$$\begin{bmatrix} N \end{bmatrix} = \operatorname{ret} N$$
$$\begin{bmatrix} x \end{bmatrix} = \operatorname{ret} x$$
$$\begin{bmatrix} \lambda x.a \end{bmatrix} = \operatorname{ret} (\lambda x.\llbracket a \rrbracket)$$
$$\begin{bmatrix} \operatorname{let} x = a \text{ in } b \end{bmatrix} = \operatorname{bind} \llbracket a \rrbracket (\lambda x.\llbracket b \rrbracket)$$
$$\begin{bmatrix} a b \rrbracket = \operatorname{bind} \llbracket a \rrbracket (\lambda v_a. \operatorname{bind} \llbracket b \rrbracket (\lambda v_b. v_a v_b))$$

These translation rules are shared between all monads.

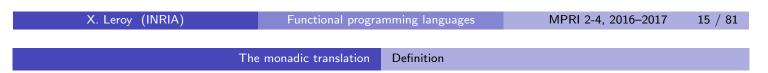
Effect on types: if  $a : \tau$  then  $\llbracket a \rrbracket : \llbracket \tau \rrbracket$  mon where  $\llbracket \tau_1 \to \tau_2 \rrbracket = \llbracket \tau_1 \rrbracket \to \llbracket \tau_2 \rrbracket$  mon and  $\llbracket \tau \rrbracket = \tau$  for base types  $\tau$ .

# The monadic translation

Extensions

$$\begin{bmatrix} \mu f.\lambda x.a \end{bmatrix} = \operatorname{ret} (\mu f.\lambda x.\llbracket a \rrbracket) \\ \begin{bmatrix} a & op & b \end{bmatrix} = \operatorname{bind} \llbracket a \rrbracket (\lambda v_a. \operatorname{bind} \llbracket b \rrbracket (\lambda v_b. \operatorname{ret} (v_a & op & v_b))) \\ \begin{bmatrix} C(a_1, \ldots, a_n) \end{bmatrix} = \operatorname{bind} \llbracket a_1 \rrbracket (\lambda v_1. \ldots \\ \operatorname{bind} \llbracket a_n \rrbracket (\lambda v_n. \operatorname{ret}(C(v_1, \ldots, v_n))))$$

$$\begin{bmatrix} \text{match } a \text{ with } \dots p_i \dots \end{bmatrix} = \text{bind } \begin{bmatrix} a \end{bmatrix} (\lambda v_a, \text{ match } v_a \text{ with } \dots \begin{bmatrix} p_i \end{bmatrix} \dots) \\ \begin{bmatrix} C(x_1, \dots, x_n) \to a \end{bmatrix} = C(x_1, \dots, x_n) \to \llbracket a \end{bmatrix}$$



# Example of monadic translation

```
 \begin{bmatrix} 1 + f & x \end{bmatrix} = \\ \text{bind (ret 1) } (\lambda v1. \\ \text{bind (bind (ret f) } (\lambda v2. \\ & \text{bind (ret x) } (\lambda v3. v2 v3))) (\lambda v4. \\ \text{ret } (v1 + v4)))
```

After administrative reductions using the first monadic law:

[[1 + f x]] =bind (f x) ( $\lambda v$ . ret (1 + v))

# Example of monadic translation

```
\llbracket \mu \text{fact. } \lambda \text{n. if n = 0 then 1 else n * fact(n-1)} \rrbracket =
      ret (\mufact. \lambdan.
               if n = 0
               then ret 1
               else bind (fact(n-1)) (\lambdav. ret (n * v))
       X. Leroy (INRIA)
                                                                         MPRI 2-4, 2016-2017
                                   Functional programming languages
                                                                                              17 / 81
                             The monadic translation
                                                   Definition
```

# The monadic translation

Monad-specific constructs and operations

Most additional constructs for exceptions, state and continuations can be treated as regular function applications of the corresponding additional operations of the monad. For instance, in the case of raise *a*:

$$\llbracket \texttt{raise } a \rrbracket = \texttt{bind (ret raise)} (\lambda v_r.\texttt{bind } \llbracket a \rrbracket (\lambda v_a. v_r v_a)) \\ \xrightarrow{adm} \texttt{bind } \llbracket a \rrbracket (\lambda v_a. \texttt{raise } v_a)$$

The bind takes care of propagating exceptions raised in a.

The only case where we need a special translation rule is the the try...with construct:

$$\llbracket \texttt{try } a \texttt{ with } x \to b \rrbracket = \texttt{trywith } \llbracket a \rrbracket (\lambda x.\llbracket b \rrbracket)$$

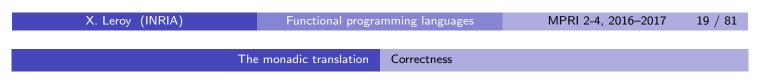
# Syntactic properties of the monadic translation

Define the monadic translation of a value  $\llbracket v \rrbracket_v$  as follows:

$$\llbracket N \rrbracket_{v} = N \qquad \llbracket \lambda x.a \rrbracket_{v} = \lambda x.\llbracket a \rrbracket$$



Lemma 2 (Monadic substitution)  $\llbracket a[x \leftarrow v] \rrbracket = \llbracket a \rrbracket [x \leftarrow \llbracket v \rrbracket_v] \text{ for all values } v,$ 



# Reasoning about reductions of the translations

If a reduces, is it the case that the translation [a] reduces? This depends on the monad:

- For the exception monad, this is true.
- For the state and continuation monads, [a] is a  $\lambda$ -abstraction which cannot reduce.

To reason about the evaluation of [a], we need in general to put this term in an appropriate context, for instance

- For the state monad: **[***a***]** *s* where *s* is a store value.
- For the continuation monad: [a] k where k is a continuation  $\lambda x \dots$

#### Contextual equivalence

To overcome this problem, we assume that the monad defines an equivalence relation  $a \approx a'$  between terms, which is reflexive, symmetric and transitive, and satisfies the following properties:

1 $(\lambda x.a) \ v \approx a[x \leftarrow v]$  $(\beta_v \text{ reduction})$ 2bind (ret v)  $(\lambda x.b) \approx b[x \leftarrow v]$ (first monadic law)3bind  $a \ (\lambda x.b) \approx \text{bind } a' \ (\lambda x.b) \text{ if } a \approx a'$ (compat. context)4If  $a \approx \text{ret } v$ , then run  $a \stackrel{*}{\rightarrow} v$ .



If  $a \Rightarrow v$ , then  $\llbracket a \rrbracket \approx \operatorname{ret} \llbracket v \rrbracket_{v}$ .

The proof is by induction on a derivation of  $a \Rightarrow v$  and case analysis on the last evaluation rule.

The cases a = N, a = x and  $a = \lambda x.b$  are obvious: we have a = v, therefore  $[\![a]\!] = \text{ret} [\![v]\!]_v$ .

# Correctness of the monadic translation

For the let case:

$$\frac{b \Rightarrow v' \quad c[x \leftarrow v'] \Rightarrow v}{\texttt{let } x = b \texttt{ in } c \Rightarrow v}$$

The following equivalences hold:

$$\begin{bmatrix} a \end{bmatrix} = \operatorname{bind} \begin{bmatrix} b \end{bmatrix} (\lambda x. \llbracket c \rrbracket)$$
  
(ind.hyp + prop.3)  $\approx$  bind (ret  $\llbracket v' \rrbracket_v$ ) ( $\lambda x. \llbracket c \rrbracket$ )  
(prop.2)  $\approx$   $\llbracket c \rrbracket [x \leftarrow \llbracket v' \rrbracket_v] = \llbracket c [x \leftarrow v'] \rrbracket$   
(ind.hyp.)  $\approx$  ret  $\llbracket v \rrbracket_v$ 

X. Leroy (INRIA)	Functional progra	mming languages	MPRI 2-4, 2016–2017	23 / 81
Tł	ne monadic translation	Correctness		

# Correctness of the monadic translation

For the application case:

$$\frac{b \Rightarrow \lambda x.d \quad c \Rightarrow v' \quad d[x \leftarrow v'] \Rightarrow v}{b \ c \Rightarrow v}$$

The following equivalences hold:

$$\begin{bmatrix} a \end{bmatrix} = \operatorname{bind} \begin{bmatrix} b \end{bmatrix} (\lambda y.\operatorname{bind} \begin{bmatrix} c \end{bmatrix} (\lambda z. y z))$$

$$(\operatorname{ind.hyp} + \operatorname{prop.3}) \approx \operatorname{bind} (\operatorname{ret} (\lambda x. \llbracket d \rrbracket)) (\lambda y.\operatorname{bind} \llbracket c \rrbracket (\lambda z. y z))$$

$$(\operatorname{prop.2}) \approx \operatorname{bind} \llbracket c \rrbracket (\lambda z. (\lambda x. \llbracket d \rrbracket) z))$$

$$(\operatorname{ind.hyp} + \operatorname{prop.3}) \approx \operatorname{bind} (\operatorname{ret} \llbracket v' \rrbracket_v (\lambda z. (\lambda x. \llbracket d \rrbracket) z))$$

$$(\operatorname{prop.2}) \approx (\lambda x. \llbracket d \rrbracket) \llbracket v' \rrbracket_v$$

$$(\operatorname{prop.1}) \approx \llbracket d \rrbracket [x \leftarrow \llbracket v' \rrbracket_v] = \llbracket d [x \leftarrow v] \rrbracket$$

$$(\operatorname{ind.hyp.}) \approx \operatorname{ret} \llbracket v \rrbracket_v$$

# Correctness of the monadic translation

Theorem 4

If  $a \Rightarrow v$ , then run  $\llbracket a \rrbracket \stackrel{*}{\rightarrow} \llbracket v \rrbracket_{v}$ .

#### Proof.

Follows from theorem 3 and property 4 of  $\approx$ .

Note that we proved this theorem only for pure terms a that do not use monad-specific constructs. These constructs add more cases, but often the proof cases for application, etc, are unchanged. (Exercise.)

X. Leroy (INRIA)	Functional programming languages		MPRI 2-4, 2016–2017	25 / 81
Т	he monadic translation	Application to some	monads	
			monaus	

# Application to the Exception monad

Define  $a_1 \approx a_2$  as  $\exists a, a_1 \xrightarrow{*} a \xleftarrow{*} a_2$ .

Some interesting properties of this relation:

- If  $a \rightarrow a'$  then  $a \approx a'$ .
- If  $a \approx a'$  and  $a \stackrel{*}{\rightarrow} v$ , then  $a' \stackrel{*}{\rightarrow} v$ .
- It is transitive, for if  $a_1 \xrightarrow{*} a \xleftarrow{*} a_2 \xrightarrow{*} a' \xleftarrow{*} a_3$ , determinism of the  $\rightarrow$ reduction implies that either  $a \xrightarrow{*} a'$  or  $a' \xrightarrow{*} a$ . In the former case,  $a_1 \stackrel{*}{\rightarrow} a' \stackrel{*}{\leftarrow} a_3$ , and in the latter case,  $a_1 \stackrel{*}{\rightarrow} a \stackrel{*}{\leftarrow} a_3$ .
- It is compatible with reduction contexts:  $E[a_1] \approx E[a_2]$  if  $a_1 \approx a_2$  and *E* is a reduction context.

We now check that  $\approx$  satisfies the hypothesis of theorem 3.

# Application to the Exception monad

X. Leroy (INRIA)	Functional programming languages		MPRI 2-4, 2016–2017	27 / 81
Th	ne monadic translation	Application to some	monads	

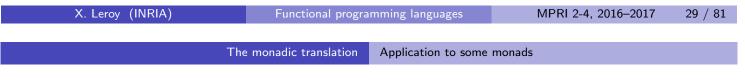
# Application to the Continuation monad

Define  $a_1 \approx a_2$  as  $\forall k \in Values, \exists a, a_1 k \xrightarrow{*} a \xleftarrow{*} a_2 k$ .

**1** 
$$(\lambda x.a) \ v \approx a[x \leftarrow v]$$
  
Trivial since  $(\lambda x.a) \ v \ k \rightarrow a[x \leftarrow v] \ k$ .
**2** bind (ret v)  $(\lambda x.b) \approx b[x \leftarrow v]$ . We have
bind (ret v)  $(\lambda x.b) \ k \rightarrow bind (\lambda k'. \ k' \ v) (\lambda x.b)$ 
 $\stackrel{*}{\rightarrow} (\lambda k'. \ k' \ v) (\lambda y. (\lambda x.b) \ y \ k)$ 
 $\rightarrow (\lambda y. (\lambda x.b) \ y \ k) \ v$ 
 $\rightarrow (\lambda x.b) \ v \ k)$ 
 $\rightarrow b[x \leftarrow v] \ k$ 

## Application to the Continuation monad

bind a<sub>1</sub> (λx.b) ≈ bind a<sub>2</sub> (λx.b) if a<sub>1</sub> ≈ a<sub>2</sub> We have bind a<sub>i</sub> (λx.b) k <sup>\*</sup>→ a<sub>i</sub> (λv. (λx.b) v k) for i = 1,2. Using the hypothesis a<sub>1</sub> ≈ a<sub>2</sub> with the continuation (λv. (λx.b) v k), we obtain a term a such that a<sub>i</sub> (λv. (λx.b) v k) <sup>\*</sup>→ a for i = 1,2. Therefore, bind a<sub>i</sub> (λx.b) k <sup>\*</sup>→ a for i = 1,2, and the result follows.
If a ≈ ret v, then run a <sup>\*</sup>→ v. The result follows from ret v (λx.x) <sup>\*</sup>→ v.

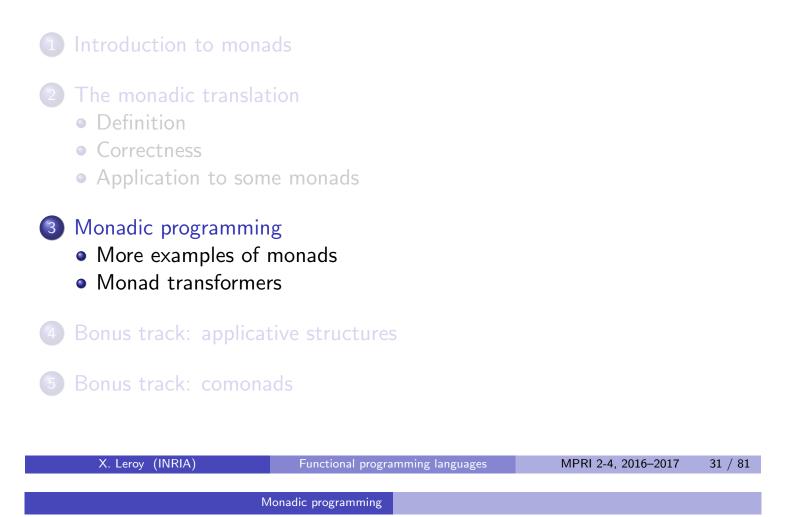


Application to the State monad

Define  $a_1 \approx a_2$  as  $\forall s \in Values, \exists a, a_1 s \xrightarrow{*} a \xleftarrow{*} a_2 s$ .

The proofs of hypotheses 1–4 are similar to those for exceptions.

# Outline



# Monads as a general programming technique

Monads provide a systematic way to structure programs into two well-separated parts:

- the algorithms proper, and
- the "plumbing" of computations needed by these algorithms (state passing, exception handling, non-deterministic choice, etc).
   The "plumbing" can often be hidden inside a reusable library.

In addition, monads can also be used to modularize code and offer new possibilities for reuse:

- Code in monadic form can be parameterized over a monad and reused with several monads.
- Monads themselves can be built in an incremental manner.

# The Counting monad (a.k.a. the Complexity monad)

Counts how many times the tick monadic operation is evaluated during execution. A special case of the State monad, with only one integer reference that can only be incremented.

```
module Count = struct

type \alpha mon = int \rightarrow \alpha \times int

let ret a = fun n -> (a, n)

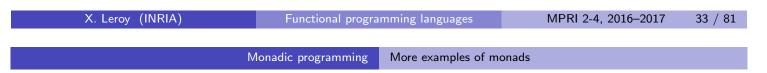
let bind m f = fun n -> match m n with (x, n') -> f x n'

let run m = m 0

let tick m = fun n -> m (n+1)

end
```

```
Infix notation: m >>= f for bind m f.
```



#### Example of use

Before monadic translation: (counts the number of comparisons)

```
let rec insert x l =
  match l with
  [] -> [x]
  | h :: t -> if tick(x < h) then x :: l else h :: insert x t</pre>
```

After monadic translation:

## The Logging monad (a.k.a. the Writer monad)

Enables computations to log messages. A generalization of the Counting monad, where a list of messages is maintained instead of a counter.

```
module Log = struct

type log = string list

type \alpha mon = log \rightarrow \alpha \times \log

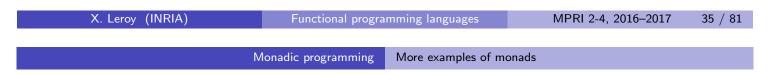
let ret a = fun l -> (a, l)

let bind m f = fun l -> match m l with (x, l') -> f x l'

let run m = match m [] with (x, l) -> (x, List.rev l)

let log msg = fun l -> ((), msg :: l)

end
```



## Example of use

Before monadic translation:

```
let abs n =
    if n >= 0
    then (log "positive"; n)
    else (log "negative"; -n)
```

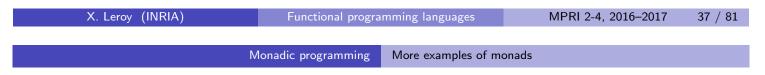
After monadic translation:

```
let abs n =
    if n >= 0
    then Log.(log "positive" >>= fun _ -> ret n)
    else Log.(log "negative" >>= fun _ -> ret (-n))
```

# The Environment monad (a.k.a. the Reader monad)

Propagate an environment "down" all branches of a computation.

```
module Environment = struct
type a mon = env -> a
let ret x = fun e -> x
let bind m f = fun e -> f (m e) e
let run m = m initial_env
let getenv varname =
  fun e -> map_lookup varname e
end
```



## Non-determinism, a.k.a. the List monad

Provides computations with non-deterministic choice as well as failure. Underneath, computes the list of all possible results.

```
module Nondet = struct
type \(\alpha\) mon = \(\alpha\) list
let ret a = a :: []
let rec bind m f =
    match m with [] -> [] | hd :: tl -> f hd @ bind tl f
let run m = match m with hd :: tl -> hd
let runall m = m
let fail = []
let either a b = a @ b
end
```

#### Example of use

All possible ways to insert an element x in a list 1:

```
let rec insert x l =
  Nondet.(either
   (ret (x :: 1))
   (match l with
        | [] -> fail
        | hd :: tl -> insert x tl >>= fun l' -> ret (hd :: l')))
```

All permutations of a list 1:

```
let rec permut l =
  match l with
  [] -> Nondet.ret []
  | hd :: tl -> Nondet.(permut tl >>= fun l' -> insert hd l')
```

X. Leroy (INRIA)	Functional programming languages		MPRI 2-4, 2016–2017	39 / 81
	Monadic programming	More examples of mo	onads	

# The Parsing monad

A variant of the state monad where the state is the input text that remains to be parsed. Supports failure like the Exception monad.

```
module Parsing = struct

type \alpha result =

| Success of \alpha * char list

| Failure

type \alpha mon = char list -> \alpha result

let ret (x: \alpha): \alpha mon = fun txt -> Success(x, txt)

let bind (m: \alpha mon) (f: \alpha -> \beta mon): \beta mon =

fun txt ->

match m txt with

| Failure -> Failure

| Success(x, txt') -> f x txt'
```

## The Parsing monad

Specific operations in this monad: symbol c (recognize and consume the single character c) and either  $m_1 m_2$  (alternative with backtracking).

```
let symbol c : char mon =
fun txt ->
match txt with
| [] -> Failure
| c' :: txt' -> if c' = c then Success(c, txt') else Failure
let either (m1: α mon) (m2: α mon): α mon =
fun txt ->
match m1 txt with
| Failure -> m2 txt
| Success(x, txt') as res -> res
```

X. Leroy (INRIA)	Functional programming languages		MPRI 2-4, 2016–2017	41 / 81
	Monadic programming	More examples of mon	ads	

# The Parsing monad

Some derived operations in this monad: 0 or 1 (opt), 0 or 1 or several (star), and 1 or several (plus) occurrences of a given recognizer m.

```
let opt (m: α mon): α option mon =
   either (m >>= fun x -> ret (Some x)) (ret None)
let rec star (m: α mon): α list mon =
   either (plus m) (ret [])
and plus (m: α mon): α list mon =
   m >>= fun x ->
   star m >>= fun y ->
   ret (x :: y)
```

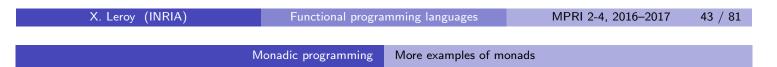
#### Monads for randomized computations

Consider a source language with randomized constructs such as

rand <i>n</i>	return a uniformly-distributed integer in [0, n[
choose <i>p a b</i>	evaluate either $a$ with probability $p \in [0,1]$ or $b$ with probability $1-p$

In a monadic interpretation, these constructs have type

rand	:	$ ext{int}  o  ext{int}  ext{mon}$	
choose	:	$\forall \alpha. \ \texttt{float} \rightarrow \alpha \ \texttt{mon} \rightarrow \alpha \ \texttt{m}$	$\mathtt{ion} \to \alpha  \mathtt{mon}$



# Examples of randomized computations

```
let dice num_sides =
    M.(rand num_sides >>= fun n -> ret (n + 1))
let roll_3d6 =
    M.(dice 6 >>= fun d1 ->
        dice 6 >>= fun d2 ->
        dice 6 >>= fun d3 ->
        ret (d1 + d2 + d3))
let traffic_light =
    M.choose 0.05 (M.ret Yellow)
                    (M.choose 0.5 (M.ret Red)
                         (M.ret Green))
```

## First implementation: the Simulation monad

Uses a pseudo-random number generator to give values to random variables (Monte-Carlo simulation). This is a variant of the State monad.

```
module Random_Simulation = struct

type \alpha mon = int \rightarrow \alpha \times int

let ret a = fun s -> (a, s)

let bind m f = fun s -> match m s with (x, s) -> f x s

let next_state s = s * 25173 + 1725

let rand n = fun s -> ((abs s) mod n, next_state s)

let choose p a b = fun s ->

if float (abs s) <= p *. float max_int

then a (next_state s) else b (next_state s)

end
```

X. Leroy (INRIA)	Functional progra	mming languages	MPRI 2-4, 2016–2017	45 / 81
N	lonadic programming	More examples of mor	nads	

# Second implementation: the Distribution monad

With the same interface, this monad computes the distribution of the results: all possible result values along with their probabilities. This is an extension of the List monad.

```
module Random_Distribution = struct
type \alpha mon = (\alpha \times float) list
let ret a = [(a, 1.0)]
let bind m f =
  [(y, p1 *. p2) | (x, p1) <- m, (y, p2) <- f x ]
let rand n = [(0, \frac{1}{n}); ...; (n-1, \frac{1}{n}) ]
let choose p a b =
  [(x, p *. p1) | (x, p1) <- a] @
  [(x, (1.0 -. p) *. p2) | (x, p2) <- b]</pre>
```

end

## Third implementation: the Expectation monad

Still with the same interface, this monad computes the expectation of a result (of type  $\alpha$ ) w.r.t. a given measure (a function  $\alpha \rightarrow \texttt{float}$ ). This is an extension of the Continuation monad.

```
module Random_Expectation = struct

type \alpha mon = (\alpha -> float) -> float

let ret x = fun k -> k x

let bind x f = fun k -> x (fun vx -> f vx k)

let rand n = fun k -> \frac{1}{n} *. k 0 +. ... +. \frac{1}{n} *. k (n-1)

let choose p a b = fun k -> p *. a k +. (1.0 -. p) *. b k

end
```

X. Leroy (INRIA)	Functional programming languages		MPRI 2-4, 2016–2017	47 / 81
	Monadic programming	Monad transformers		

# Combining monads

What if we need both exceptions and state in an algorithm? We can write (from scratch) a monad that supports both. Notice that there are several choices:

• type  $\alpha \mod = \texttt{state} \rightarrow (\alpha \times \texttt{state})$  outcome I.e. the state is discarded when we raise an exception.

• type  $\alpha \mod = \texttt{state} \rightarrow \alpha \texttt{ outcome} \times \texttt{state}$ 

I.e. the state is kept when we raise an exception.

In the second case, trywith can be defined in two ways:

The *s* choice backtracks the assignments made by the computation m; the s' choice preserves them.

## Composing two monads?

Given two monads

```
\begin{array}{ll} \texttt{type } \alpha \texttt{ mon1} & \texttt{type } \alpha \texttt{ mon2} \\ \texttt{ret1}: \alpha \to \alpha \texttt{ mon1} & \texttt{ret2}: \alpha \to \alpha \texttt{ mon2} \\ \texttt{bind1}: \alpha \texttt{ mon1} \to (\alpha \to \beta \texttt{ mon1}) & \texttt{bind2}: \alpha \texttt{ mon2} \to (\alpha \to \beta \texttt{ mon2}) \\ \to \beta \texttt{ mon1} & \to \beta \texttt{ mon2} \end{array}
```

is there a generic way to compose them? Let's try...

type  $\alpha \mod = \alpha \mod 1 \mod 2$ let ret  $(x : \alpha) : \alpha \mod = \operatorname{ret2} (\operatorname{ret1} x)$ let bind  $(x : \alpha \mod) (f : \alpha \rightarrow \beta \mod) : \beta \mod =$ bind2  $x (\lambda y : \alpha \mod 1. \operatorname{ret2} ($ bind1  $y (\lambda z : \alpha. ???? (f z))))$ 

X. Leroy (INRIA)	Functional programming languages		MPRI 2-4, 2016–2017	49 / 81
	Monadic programming	Monad transformers		

Composing two monads?

Without additional operations provided by the second (outer) monad, there is no way to define the bind of the composed monad:

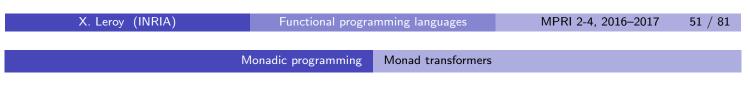
let bind  $(x : \alpha \text{ mon})$   $(f : \alpha \rightarrow \beta \text{ mon}) : \beta \text{ mon} =$ bind2 x  $(\lambda y : \alpha \text{ mon1. ret2} ($ bind1 y  $(\lambda z : \alpha. ???? (f z))))$ 

Since  $f \ z : \beta \mod 1 \mod 2$  and bind1 demands something of type  $\beta \mod 1$ , we need a term ???? of type  $\beta \mod 1 \mod 2 \rightarrow \beta \mod 1$ .

It is impossible to construct a closed, terminating term of this type just from the ret2 and bind2 operations of the second monad!

A monad transformer takes any monad M and returns a monad M' with additional capabilities, e.g. exceptions, state, continuation. It also provides a lift function that transforms M computations (of type  $\alpha$  M.mon) into M' computations (of type  $\alpha$  M'.mon)

In Caml, monad transformers are naturally presented as functors, i.e. functions from modules to modules. (Haskell uses type classes.)



# Signature for monads

The Caml module signature for a monad is:

```
module type MONAD = sig

type \alpha mon

val ret: \alpha \rightarrow \alpha mon

val bind: \alpha mon \rightarrow (\alpha \rightarrow \beta \text{ mon}) \rightarrow \beta mon

val run: \alpha mon \rightarrow \alpha

end
```

## The Identity monad

The Identity monad is a trivial instance of this signature:

```
module Identity = struct
    type \alpha mon = \alpha
    let ret x = x
    let bind m f = f m
    let run m = m
```

end

X. Leroy (INRIA)	Functional programming languages		MPRI 2-4, 2016–2017	53 / 81
Ν	Ionadic programming	Monad transformers		

# Monad transformer for exceptions

```
module ExceptionTransf(M: MONAD) = struct
  type \alpha outcome = V of \alpha | E of exn
  type \alpha mon = (\alpha outcome) M.mon
  let ret x = M.ret (V x)
  let bind m f = 
    M.bind m (function E e \rightarrow M.ret (E e) | V v \rightarrow f v)
  let lift x = M.bind x (fun v \rightarrow M.ret (V v))
  let run m = M.run (M.bind m (function V x -> M.ret x))
  let raise e = M.ret (E e)
  let trywith m f =
    M.bind m (function E e \rightarrow f e | V v \rightarrow M.ret (V v))
end
```

#### Monadic programming Monad transformers

#### Monad transformer for state

```
module StateTransf(M: MONAD) = struct
type a mon = state -> (a * state) M.mon
let ret x = fun s -> M.ret (x, s)
let bind m f =
  fun s -> M.bind (m s) (fun (x, s') -> f x s')
let lift m = fun s -> M.bind m (fun x -> M.ret (x, s))
let run m =
  M.run (M.bind (m empty_store) (fun (x, s') -> M.ret x))
let ref x = fun s -> M.ret (store_alloc x s)
let deref r = fun s -> M.ret (store_read r s, s)
let assign r x = fun s -> M.ret (store_write r x s)
end
```

```
X. Leroy (INRIA) Functional programming languages MPRI 2-4, 2016–2017 55 / 81

Monadic programming Monad transformers
```

Monad transformer for continuations

```
module ContTransf(M: MONAD) = struct
type \alpha mon = (\alpha -> answer M.mon) -> answer M.mon
let ret x = fun k -> k x
let bind m f = fun k -> m (fun v -> f v k)
let lift m = fun k -> M.bind m k
let run m = M.run (m (fun x -> M.ret x))
let callcc f = fun k -> f k k
let throw c x = fun k -> c x
end
```

#### Using monad transformers

```
module StateAndException = struct
    include ExceptionTransf(State)
    let ref x = lift (State.ref x)
    let deref r = lift (State.deref r)
    let assign r x = lift (State.assign r x)
    end
```

This gives a type  $\alpha \mod = \texttt{state} \rightarrow \alpha \texttt{ outcome} \times \texttt{state}$ , i.e. state is preserved when raising exceptions.

```
The other combination, StateTransf(Exception) gives \alpha \mod = \texttt{state} \rightarrow (\alpha \times \texttt{state}) \texttt{outcome},
i.e. state is discarded when an exception is raised.
```



The Concurrency monad transformer (Based on an approach by Tomas Petricek, 2011)

Given any monad M, we define concurrency (interleaving of computations) via the following type of resumptions:

```
module Concur(M: MONAD) = struct

type \alpha mon =

| Done of \alpha

| Step of (\alpha mon) M.mon
```

A resumption describes a sequence of computations in monad M:

- Done v denotes no computations and a final result value v
- Step *m* denotes the computation *m* followed by the resumption that *m* returns as its value.

## The Concurrency monad transformer

type  $\alpha$  mon = Done of  $\alpha$  | Step of ( $\alpha$  mon) M.mon

A resumption can trivially be turned into a single computation in monad M, then run:

```
let rec perform (x: α mon): α M.mon =
  match x with
  | Done res -> M.ret res
  | Step m -> M.bind m perform
let run (x: α mon) = M.run (perform x)
```

However, by keeping the list-like structure of resumptions, we are able to interleave two resumptions, simulating concurrent execution.

X. Leroy (INRIA)	Functional progra	mming languages	MPRI 2-4, 2016–2017	59 / 81
I	Monadic programming	Monad transformers		

#### The Concurrency monad transformer

The ret operation of the Concurrency monad performs zero computations:

let ret (x:  $\alpha$ ):  $\alpha$  mon = Done x

The act operation (also known as lift) performs just one computation:

let act (m: \alpha M.mon): \alpha mon =
 Step (M.bind m (fun res -> M.ret (Done res)))

The bind operation is similar to list concatenation, appending two lists of computations:

```
let rec bind (m: α mon) (f: α -> β mon): β mon =
   match m with
   | Done res -> f res
   | Step s -> Step (M.bind s (fun m' -> M.ret (bind m' f)))
```

#### Monadic programming Monad transformers

#### The Concurrency monad transformer

Finally, par interleaves the computations of two resumptions:

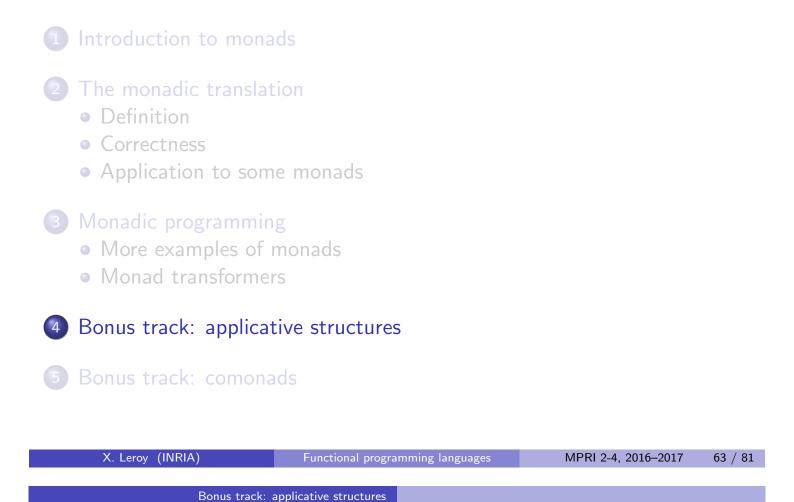


#### Example of use

```
module M = Concur(Log)
let rec loop n s =
    if n <= 0
    then M.ret ()
    else M.(act (Log.log s) >>= fun _ -> loop (n-1) s)
M.(run (act (Log.log "start:") >>= fun _ ->
        par (loop 6 "a") (loop 4 "b")))
```

This code will log "start:ababababa"

## Outline



# Applicative structures

Monads impose a very "sequential" programming style, where all subcomputations are explicitly sequenced. This is heavy style if the monad has no effects or effects that commute (e.g. read effects).

Example: evaluating expressions containing variables using the Environment monad.

```
In direct style:
In monadic style:
let rec eval = function
| Const n -> n
| Var v -> getenv v
| Plus(a, b) ->
eval a + eval b
In monadic style:
let rec eval = function
| Const n -> ret n
| Var v -> getenv v
| Plus(a, b) ->
eval a + eval b
ret (n + m)
```

# Monadic application combinators

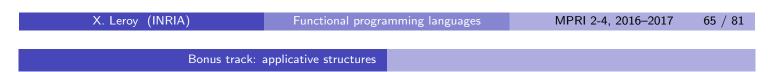
This can be improved by defining application combinators:

```
let mapp (f: \alpha \rightarrow \beta) (m: \alpha mon) : \beta mon = bind m (fun x -> ret (f x))
```

let mapp2 (f:  $\alpha \rightarrow \beta \rightarrow \gamma$ ) (m1:  $\alpha$  mon) (m2:  $\beta$  mon):  $\gamma$  mon = bind m1 (fun x1 -> bind m2 (fun x2 -> ret (f x1 x2)))

(etc). The eval function becomes nicer:

```
let rec eval = function
...
| Plus(a, b) -> mapp2 (+) (eval a) (eval b)
```



# The monadic application combinator

We can avoid defining a combinator for every arity, as follows:

```
let <*> (f: (\alpha \rightarrow \beta) mon) (m: \alpha mon) : \beta mon =
bind f (fun vf -> bind m (fun vm -> ret (vf vm)))
```

Using the fact that <\*> associates to the left in OCaml, we get:

```
let rec eval = function
...
| Plus(a, b) -> ret (+) <*> eval a <*> eval b
```

More generally: ret  $f \ll m_1 \ll m_n$ denotes the application of the pure function f to the results of the monadic computations  $m_1, \ldots, m_n$ .

#### Applicative structures

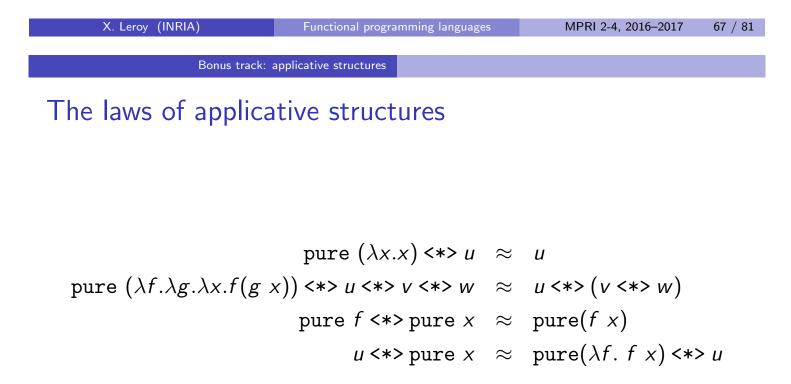
(C. McBride and R. Paterson, Applicative structures with effects, JFP 18(1), 2008)

An applicative structure is a parameterized type  $\alpha$  app of effectful computations producing a value of type  $\alpha$ , plus two operations:

pure :  $\forall \alpha, \ \alpha \to \alpha \text{ app}$ <\*> :  $\forall \alpha \beta, \ (\alpha \to \beta) \text{ app} \to \alpha \text{ app} \to \beta \text{ app}$ 

pure embeds values (computations without effects) into computations.

<\*>, pronounced "apply", performs function application with propagation of effects.



Intuitively: we can reorder/simplify pure computations, as long as the order of effectful computations is preserved.

Categorically: a strong lax monoidal functor...

#### Monads and applicative structures

Every monad M defines an applicative structure:

```
type \alpha app = \alpha M.mon
let pure x = M.ret x
let <*> f x =
    M.bind f (fun vf -> M.bind x (fun vx -> M.ret (vf vx)))
```

(This is for left-to-right application. Can also do right-to-left by swapping the two bind.)

However:

- Sometimes, other definitions of <\*> are more useful.
- Some types are not monads but have a useful applicative structure.

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4, 2016–2017	69 / 81
Bonus track: a	applicative structures		

# Monads with nonstandard application

Consider the Exception monad. We'd like to collect all the exceptions raised during the evaluation of an expression, not just the first one:

(raise A) + (raise B) ---> Uncaught exceptions: A, B

Let us redefine the type of the monad as

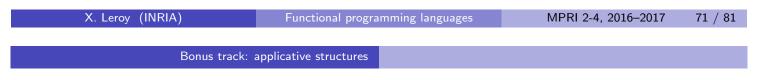
```
type \alpha mon = V of \alpha | E of exn list
let raise e = E [e]
let bind m f =
   match m with
   | V x -> f x
   | E exnlist -> E exnlist
```

For bind m f, if m raises an exception, we have no value to pass to f, so we cannot collect the exceptions raised by f. (Unavoidable!)

# Monads with nonstandard application

Rather than defining <\*> in terms of bind, which will not collect all exceptions, we provide a more useful definition:

```
let <*> f x =
    match f, x with
    | V vf, V vx -> V (vf vx)
    | V vf, E ex -> E ex
    | E ef, V vx -> E ef
    | E ef, E ex -> E (ef @ ex) (* list concatenation *)
Thus, ret (+) <*> raise A <*> raise B
produces E [A; B], as desired.
```



# Applicative structures without bind

For some monads, we would like to compute both

- static information on the structure of the computation, and
- a dynamic interpretation that actually performs the computation, perhaps using the static information to be more efficient.

Simple example: for arithmetic expressions with variables, compute the free variables as static info and the function *environment*  $\rightarrow$  *value* as dynamic interpretation.

More realistic example: for parsing combinators, compute *nullable* and *first* information on the parsers as static info, and the function  $text \rightarrow result \times text$  as dynamic interpretation; use the static info to quickly eliminate impossible cases in the *either* combinator.

#### Free variables in the Environment monad

Let us try to extend the Environment monad with the computation of free variables.

```
type lpha mon = stringset * (env -> lpha)
```

(static information  $\times$  dynamic interpretation)

```
let ret x = (emptyset, fun e -> x)
let getenv v = (singleton v, fun e -> map_lookup v e)
let <*> (sf,df) (sx,dx) =
   (union sf sx, fun e -> df (dx e) e)
```

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4, 2016–2017	73 / 81
Bonus track:	applicative structures		

#### Free variables in the Environment monad

However, bind cannot be defined!

```
type \alpha mon = stringset * (env -> \alpha)
```

```
let bind (sx,dx : \alpha mon) (f: \alpha \rightarrow \beta mon) : \beta mon = (union sx (fst (f ???)), fun e \rightarrow snd (f (dx e)))
```

When computing the static part of the result, we do not have any value to pass to function f so that we can extract the static part of f's result!

(Besides: the static part of f's result can depend on the value being passed to f.)

 $\rightarrow$  This extended Environment is an applicative structure that is not a monad.

# Composing applicative structures

Another evidence that applicative structures differ from monads is that applicative structures compose naturally: given

type $lpha$ app1	type $lpha$ app2
pure1 : $lpha  ightarrow lpha$ app1	pure2 : $lpha  ightarrow lpha$ app2
<*>1 : ( $lpha  ightarrow eta$ ) $ ightarrow$	<*>2 : ( $\alpha \rightarrow \beta$ ) $\rightarrow$
lpha app1 $ ightarrow eta$ app1	lpha app2 $ ightarrow eta$ app2

we can define

type 
$$\alpha$$
 app =  $\alpha$  app1 app2  
let pure  $(x : \alpha) : \alpha$  app = pure2 (pure1 x)  
let <\*>  $(f : \alpha \rightarrow \beta) : \alpha$  app  $\rightarrow \beta$  app = <\*>2 (<\*>1 f)

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4, 2016–2017	75 / 81
	Bonus track: comonads		
Outline			
1 Introduction to mor	nads		
<ul> <li>2 The monadic translation</li> <li>• Definition</li> <li>• Correctness</li> <li>• Application to so</li> </ul>			
<ul> <li>Monadic programm</li> <li>More examples of</li> <li>Monad transform</li> </ul>	f monads		
④ Bonus track: applic	ative structures		
5 Bonus track: comor	nads		

#### Comonads

The categorical dual of monads (what else?)

A comonad is defined by a parameterized type  $\alpha$  com and operations proj and cobind, with types:

```
proj : \forall \alpha. \ \alpha \ \operatorname{com} \rightarrow \alpha
cobind : \forall \alpha, \beta. \ (\alpha \ \operatorname{com} \rightarrow \beta) \rightarrow \alpha \ \operatorname{com} \rightarrow \beta \ \operatorname{com}
```

The type  $\tau$  com is the type of processes that produce values of type  $\tau$ . (For example: a collection of  $\tau$  values.)

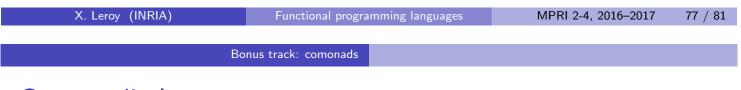
proj *a* extracts a value from such a process *a*.

cobind f a, given

– a function f that produces a  $\beta$  value from a  $\alpha$  com process,

– and a  $\alpha$  com process,

extends function f to construct a process producing  $\beta$ 's.



Comonadic laws

The proj and cobind operations of the comonad are supposed to satisfy the following algebraic laws:

> proj(cobind k x)  $\approx k x$ cobind proj  $x \approx x$ cobind( $k_2 \circ \text{cobind } k_1$ )  $\approx$  cobind  $k_2 \circ \text{cobind } k_1$

```
\begin{aligned} \text{Lazy evaluation as a comonad} \\ \text{module Lazy = struct} \\ \text{type } \alpha \text{ com } = \alpha \text{ status ref} \\ \text{and } \alpha \text{ status } = \\ & | \text{ Evaluated of } \alpha \\ & | \text{ Suspended of unit } \neg \rangle \alpha \\ \text{let proj } (x: \alpha \text{ com}): \alpha = \\ & \text{match } !x \text{ with} \\ & | \text{ Evaluated } v \neg \rangle v \\ & | \text{ Suspended } f \neg \rangle \text{ let } v = f() \text{ in } x := \text{Evaluated } v; v \\ & \text{let cobind } (f: \alpha \text{ com } \neg \beta) (x: \alpha \text{ com}) : \beta \text{ com } = \\ & \text{ref } (\text{Suspended } (\text{fun } () \neg \rangle \text{ f } x)) \end{aligned}
```

X. Leroy (INRIA)	Functional programming languages	MPRI 2-4, 2016–2017	79 / 81
В			

# Lazy evaluation as a comonad

We can also equip lazy evaluation with monadic ret and monadic bind:

```
let ret (x: \alpha) : \alpha com = ref (Evaluated x)
let bind (x: \alpha com) (f: \alpha \rightarrow \beta com): \beta com = f (proj x)
```

However, if we only have ret and bind, there is no way to suspend the evaluation of a nontrivial computation: ret always evaluates its argument!

In contrast, cobind f x is equivalent to lazy(f x) in Caml and let us suspend arbitrary computations.

## Other uses of comonads

Working with infinite, lazy data structures: streams, bidirectional streams, etc. (See example with cellular automata in companion file monads.ml)

Semantics of dataflow languages and reactive languages. (Tarmo Uustalu, Varmo Vene. *The Essence of Dataflow Programming.* APLAS 2005.)

X. Leroy (INRIA)

Functional programming languages

MPRI 2-4, 2016–2017

81 / 81