Monads in programming language theory

Monads are a technical device 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.)
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:

\[
\begin{align*}
\text{[}\mathit{N}\text{]} &= \mathit{V}(N) & \text{[}\mathit{\lambda x.\ }\mathit{a}\text{]} &= \mathit{V}(\lambda x.\text{[}\mathit{a}\text{]}), \\
\text{[}\mathit{x}\text{]} &= \mathit{V}(x) & \text{[}\mathit{\lambda x.\ }\mathit{a}\text{]} &= \lambda s.(\mathit{\lambda x.\ }\mathit{a}, s), \\
\text{[}\mathit{\lambda x.\ }\mathit{a}\text{]} &= \lambda s.(\mathit{\lambda x.\ }\mathit{a}, s) & \text{[}\mathit{\lambda x.\ }\mathit{a}\text{]} &= \lambda k.k (\mathit{\lambda x.\ }\mathit{[}\mathit{a}\text{]}),
\end{align*}
\]

in all three cases, we return (put in some appropriate wrapper) the values \(N\) or \(x\) or \(\lambda x.\text{[}\mathit{a}\text{]}\).
Commonalities between program transformations

For let bindings, we have:

\[
\begin{align*}
\text{[let } x = a \text{ in } b] & = \ \text{match } [a] \text{ with } E(x) \to E(x) \mid V(x) \to [b] \\
\text{[let } x = a \text{ in } b] & = \ \lambda s. \ \text{match } [a] s \text{ with } (x, s') \to [b] s' \\
\text{[let } x = a \text{ in } b] & = \ \lambda k. \ [a] (\lambda x. \ [b] k)
\end{align*}
\]

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 \([b]\).

Concerning function applications:

\[
\begin{align*}
\text{[a b]} & = \ \text{match } [a] \text{ with } \\
& \quad | E(e_a) \to E(e_a) \\
& \quad | V(v_a) \to \\
& \quad \ \text{match } [b] \text{ with } E(e_b) \to E(e_b) \mid V(v_b) \to v_a v_b \\
\text{[a b]} & = \ \lambda s. \ \text{match } [a] s \text{ with } (v_a, s') \to \\
& \quad \ \text{match } [b] s' \text{ with } (v_b, s'') \to v_a v_b s'' \\
\text{[a b]} & = \ \lambda k. \ [a] (\lambda v_a. \ [b] (\lambda v_b. \ v_a v_b k))
\end{align*}
\]

We bind \([a]\) to a variable \(v_a\), then bind \([b]\) 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 $\alpha \text{ mon}$ and operations $\text{ret}$, $\text{bind}$ and $\text{run}$, with types:

$$\text{ret} : \forall \alpha. \alpha \to \alpha \text{ mon}$$
$$\text{bind} : \forall \alpha, \beta. \alpha \text{ mon} \to (\alpha \to \beta \text{ mon}) \to \beta \text{ mon}$$
$$\text{run} : \forall \alpha. \alpha \text{ mon} \to \alpha$$

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

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

$\text{bind} \ a \left(\lambda x. b\right)$ performs the computation $a : \tau \text{ mon}$, binds its value to $x : \tau$, then performs the computation $b : \tau' \text{ mon}$.

$\text{run} \ a$ is the execution of a whole monadic program $a$, extracting its return value.

Monadic laws

The $\text{ret}$ and $\text{bind}$ operations of the monad are supposed to satisfy the following algebraic laws:

$$\text{bind} \ (\text{ret} \ a) \ f \ \approx \ f \ a$$
$$\text{bind} \ a \left(\lambda x. \text{ret} \ x\right) \ \approx \ a$$
$$\text{bind} \ (\text{bind} \ a \left(\lambda x. b\right)) \left(\lambda y. c\right) \ \approx \ \text{bind} \ a \left(\lambda x. \text{bind} \ b \left(\lambda y. c\right)\right)$$

The relation $\approx$ needs to be made more precise, but intuitively means “behaves identically”.
Example: the Exception monad
(also called the Error monad)

\[
\text{type } \alpha \text{ mon } = \text{V of } \alpha \mid \text{E of exn}
\]
\[
\text{ret } a = \text{V}(a)
\]
\[
\text{bind } m \ f = \text{match } m \text{ with } \text{E}(x) \rightarrow \text{E}(x) \mid \text{V}(x) \rightarrow f \ x
\]
\[
\text{run } m = \text{match } m \text{ with } \text{V}(x) \rightarrow x
\]
bind encapsulates the propagation of exceptions in compound expressions such as \(a \ b\) or let bindings.

Additional operations in this monad:
\[
\text{raise } x = \text{E}(x)
\]
\[
\text{trywith } m \ f = \text{match } m \text{ with } \text{E}(x) \rightarrow f \ x \mid \text{V}(x) \rightarrow \text{V}(x)
\]

---

Example: the State monad

\[
\text{type } \alpha \text{ mon } = \text{state } \rightarrow \alpha \times \text{state}
\]
\[
\text{ret } a = \lambda s. \ (a, s)
\]
\[
\text{bind } m \ f = \lambda s. \ \text{match } m \ s \text{ with } (x, s') \rightarrow f \ x \ s'
\]
\[
\text{run } m = \text{match } m \text{ empty_store with } (x, s) \rightarrow x
\]
bind encapsulates the threading of the state in compound expressions.

Additional operations in this monad:
\[
\text{ref } x = \lambda s. \ \text{store_alloc } x \ s
\]
\[
\text{deref } r = \lambda s. \ (\text{store_read } r \ s, s)
\]
\[
\text{assign } r \ x = \lambda s. \ \text{store_write } r \ x \ s
\]
Example: the Continuation monad

\[
\text{type } \alpha \text{ mon } = (\alpha \to \text{answer}) \to \text{answer}
\]

\[
\text{ret a } = \lambda k. \ k \ a
\]

\[
\text{bind } m \ f = \lambda k. \ m \ (\lambda v. \ f \ v \ k)
\]

\[
\text{run } m = m \ (\lambda x. \ x)
\]

Additional operations in this monad:

\[
\text{callcc } f = \lambda k. \ f \ k \ k
\]

\[
\text{throw } x \ y = \lambda k. \ x \ y
\]
The monadic translation

Definition

Core constructs

\[
\begin{align*}
[N] &= \text{ret } N \\
[x] &= \text{ret } x \\
[\lambda x.a] &= \text{ret } (\lambda x.[a]) \\
[\text{let } x = a \text{ in } b] &= \text{bind } [a] (\lambda x.[b]) \\
[a \ b] &= \text{bind } [a] (\lambda v_a. \text{bind } [b] (\lambda v_b. v_a v_b))
\end{align*}
\]

These translation rules are shared between all monads.

Effect on types: if \(a : \tau\) then \([a] : [\tau]\) mon

where \([\tau_1 \rightarrow \tau_2] = \tau_1 \rightarrow \[\tau_2]\) mon and \([\tau] = \tau\) for base types \(\tau\).

Extensions

\[
\begin{align*}
[\mu f.\lambda x.a] &= \text{ret } (\mu f.\lambda x.[a]) \\
[a \ op \ b] &= \text{bind } [a] (\lambda v_a. \text{bind } [b] (\lambda v_b. \text{ret } (v_a \ op \ v_b))) \\
[C(a_1, \ldots, a_n)] &= \text{bind } [a_1] (\lambda v_1. \ldots \text{bind } [a_n] (\lambda v_n. \text{ret}(C(v_1, \ldots, v_n))))
\end{align*}
\]

\[
\begin{align*}
[\text{match } a \text{ with } \ldots p_i \ldots] &= \text{bind } [a] (\lambda v_a. \text{match } v_a \text{ with } \ldots [p_i] \ldots) \\
[C(x_1, \ldots, x_n) \rightarrow a] &= C(x_1, \ldots, x_n) \rightarrow [a]
\end{align*}
\]
Example of monadic translation

\[ [1 + f \ x] = \]
\[ \text{bind (ret 1) (} \lambda v_1. \]
\[ \text{bind (bind (ret f) (} \lambda v_2. \]
\[ \text{bind (ret x) (} \lambda v_3. v_2 \ v_3)) (} \lambda v_4. \]
\[ \text{ret (} v_1 \ + \ v_4)) \]

After administrative reductions using the first monadic law:

\[ [1 + f \ x] = \]
\[ \text{bind (} f \ x) (} \lambda v. \text{ret (} 1 \ + \ v)) \]

Example of monadic translation

\[ [\mu \text{fact. } \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact(n-1)}] = \]
\[ \text{ret (} \mu \text{fact. } \lambda n. \]
\[ \text{if } n = 0 \]
\[ \text{then ret 1} \]
\[ \text{else bind (fact(n-1)) (} \lambda v. \text{ret (} n * \ v)) \]
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`:

\[
\begin{align*}
\text{[raise } a\text{]} &= \text{bind (ret raise)} (\lambda v_r. \text{bind } [a] (\lambda v_a. v_r \ v_a)) \\
&\overset{adm}{\rightarrow} \text{bind } [a] (\lambda v_a. \text{raise } v_a)
\end{align*}
\]

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:

\[
\text{[try a with } x \rightarrow b\text{]} = \text{trywith } [a] (\lambda x. [b])
\]

Syntactic properties of the monadic translation

Define the monadic translation of a value \( [v]_v \) as follows:

\[
[N]_v = N \quad [\lambda x.a]_v = \lambda x.[a]
\]

**Lemma 1 (Translation of values)**

\( [v] = \text{ret } [v]_v \) for all values \( v \). Moreover, \( [v]_v \) is a value.

**Lemma 2 (Monadic substitution)**

\( [a[x \leftarrow v]] = [a][x \leftarrow [v]_v] \) 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 \(\lambda x\ldots\)

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. \((\lambda x.a)\ v \approx a[x \leftarrow v]\)
2. \(\text{bind } (\text{ret } v) (\lambda x.b) \approx b[x \leftarrow v]\)
3. \(\text{bind } a (\lambda x.b) \approx \text{bind } a' (\lambda x.b)\) if \(a \approx a'\)
4. If \(a \approx \text{ret } v\), then \(\text{run } a \xrightarrow{*} v\).
Correctness of the monadic translation

Theorem 3

If $a \Rightarrow v$, then $[a] \approx \text{ret } [v]_v$.

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 $[a] = \text{ret } [v]_v$.

For the let case:

\[
\begin{align*}
  b & \Rightarrow v' \\
  c[x \leftarrow v'] & \Rightarrow v \\
\text{let } x = b \text{ in } c & \Rightarrow v
\end{align*}
\]

The following equivalences hold:

\[
\begin{align*}
  [a] = \text{bind } [b] (\lambda x. [c]) \\
  \text{(ind.hyp + prop.3)} & \approx \text{bind } (\text{ret } [v']_v) (\lambda x. [c]) \\
  \text{(prop.2)} & \approx [c][x \leftarrow [v']_v] = [c[x \leftarrow v']]
\end{align*}
\]
Correctness of the monadic translation

For the application case:

\[
\begin{align*}
    b & \Rightarrow \lambda x. d \\
    c & \Rightarrow v' \\
    d[x \leftarrow v'] & \Rightarrow v
\end{align*}
\]

\[
b \ c & \Rightarrow v
\]

The following equivalences hold:

\[
[a] = \text{bind } \left[b\right] (\lambda y. \text{bind } \left[c\right] (\lambda z. y \ z))
\]

\[
\text{(ind.hyp + prop.3)} \approx \text{bind } (\text{ret } (\lambda x. \left[d\right])) (\lambda y. \text{bind } \left[c\right] (\lambda z. y \ z))
\]

\[
\text{(prop.2)} \approx \text{bind } \left[c\right] (\lambda z. (\lambda x. \left[d\right]) z)
\]

\[
\text{(ind.hyp + prop.3)} \approx \text{bind } (\text{ret } \left[v'\right]_v (\lambda z. (\lambda x. \left[d\right]) z))
\]

\[
\text{(prop.2)} \approx (\lambda x. \left[d\right]) \left[v'\right]_v
\]

\[
\text{(prop.1)} \approx \left[d\right][x \leftarrow \left[v'\right]_v] = \left[d[x \leftarrow v]\right]
\]

\[
\text{(ind.hyp.)} \approx \text{ret } \left[v\right]_v
\]

Theorem 4

\text{If } a \Rightarrow N, \text{ then } \text{ run } [a] \xrightarrow{\ast} N.

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.)
Application to the Exception monad

Define $a_1 \approx a_2$ as $\exists a, a_1 \leadsto a \leftarrow a_2$.

Some interesting properties of this relation:

- If $a \leadsto a'$ then $a \approx a'$.
- If $a \approx a'$ and $a \leftarrow v$, then $a' \leftarrow v$.
- It is transitive, for if $a_1 \leftarrow a_2 \leftarrow a' \leftarrow a_3$, determinism of the $\leadsto$ reduction implies that either $a \leftarrow a'$ or $a' \leftarrow a$. In the former case, $a_1 \leftarrow a' \leftarrow a_3$, and in the latter case, $a_1 \leftarrow a \leftarrow a_3$.
- 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.

\begin{itemize}
  \item $(\lambda x.a) v \approx a[x \leftarrow v]$
  Trivial since $(\lambda x.a) v \leadsto a[x \leftarrow v]$.
  \item bind $(\text{ret } v) (\lambda x.b) \approx b[x \leftarrow v]$. We have
    \begin{align*}
      \text{bind } (\text{ret } v) (\lambda x.b) & \leadsto \text{bind } (V(v)) (\lambda x.b) \\
      & \approx \text{match } V(v) \text{ with } E(y) \rightarrow y \mid V(z) \rightarrow (\lambda x.b) z \\
      & \leadsto (\lambda x.b) v \rightarrow b[x \leftarrow v]
    \end{align*}
  \item bind $a_1 (\lambda x.b) \approx \text{bind } a_2 (\lambda x.b)$ if $a_1 \approx a_2$.
    Trivial since $\text{bind } [] (\lambda x.b)$ is an evaluation context.
  \item If $a \approx \text{ret } v$, then $\text{run } a \approx v$.
    Since $\text{ret } v \leftarrow V(v)$, we have $a \leadsto V(v)$ and the result follows.
\end{itemize}
Application to the Continuation monad

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

1. $(\lambda x.a) v \approx a[x \leftarrow v]$  
   Trivial since $(\lambda x.a) v k \rightarrow a[x \leftarrow v] k$.

2. $\text{bind} (\text{ret } v) (\lambda x.b) \approx b[x \leftarrow v]$. We have

   \[
   \text{bind} (\text{ret } v) (\lambda x.b) \rightarrow \text{bind} (\lambda k'. k' v) (\lambda x.b)
   \]

   \[
   \rightarrow (\lambda y. (\lambda x.b) y k)
   \]

   \[
   \rightarrow (\lambda x.b) v k
   \]

   \[
   \rightarrow b[x \leftarrow v] k
   \]

Application to the Continuation monad

1. $\text{bind } a_1 (\lambda x.b) \approx \text{bind } a_2 (\lambda x.b)$ if $a_1 \approx a_2$

   We have $\text{bind } a_i (\lambda x.b) k \rightarrow a_i (\lambda v. (\lambda x.b) v k)$ for $i = 1, 2$.

   Using the hypothesis $a_1 \approx a_2$ with the continuation $(\lambda v. (\lambda x.b) v k)$, we obtain a term $a$ such that $a_i (\lambda v. (\lambda x.b) v k) \rightarrow a$ for $i = 1, 2$.

   Therefore, $\text{bind } a_i (\lambda x.b) k \rightarrow a$ for $i = 1, 2$, and the result follows.

2. If $a \approx \text{ret } v$, then $\text{run } a \rightarrow v$.

   The result follows from $\text{ret } v (\lambda x.x) \rightarrow v$. 
Application to the State monad

Define $a_1 \approx a_2$ as $\forall s \in \text{Values}, \exists a, a_1 s \rightarrow a \leftarrow a_2 s$.

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

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).

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

Enables computations to log messages. A special case of the State monad, guaranteeing that the log grows monotonically.

```
module Log = struct
  type log = string list
  type α mon = log → α × 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
```

X. Leroy (INRIA)
Example of use

Before monadic translation:

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

After monadic translation:

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

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.

```plaintext
module Nondet = struct
  type α mon = α 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 \( l \):

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

All permutations of a list \( l \):

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

Monads for randomized computations

Consider a source language with randomized constructs such as

- \( \text{rand } n \) return a uniformly-distributed integer in \([0, n]\)
- \( \text{choose } p a b \) evaluate \( a \) with probability \( p \in [0, 1] \) and \( b \) with probability \( 1 - p \)

In a monadic interpretation, these constructs have type

- \( \text{rand} : \text{int} \to \text{int mon} \)
- \( \text{choose} : \forall \alpha. \text{float} \to \alpha \text{ mon} \to \alpha \text{ mon} \to \alpha \text{ mon} \)
Examples of randomized computations

```ocaml
let roll_3d6 = M.bind (M.rand 6) (fun d1 ->
                  M.bind (M.rand 6) (fun d2 ->
                  M.bind (M.rand 6) (fun d3 ->
                          M.return (1+d1 + 1+d2 + 1+d3))))

let traffic_light =
      M.choose 0.05 (M.return Yellow)
                  (M.choose 0.5 (M.return Red)
                  (M.return 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.

```ocaml
module Random_Simulation = struct
  type α mon = int → α × 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
```
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 α mon = (α × 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, 1.0/n); ...; (n-1, 1.0/n) ]
  let choose p a b = 
    [ (x, p *. p1) | (x, p1) <- a ] @
    [ (x, (1.0 -. p) *. p2) | (x, p2) <- b ]
end
```

Third implementation: the Expectation monad

Still with the same interface, this monad computes the expectation of a result (of type α) w.r.t. a given measure (a function α → float). This is an extension of the Continuation monad.

```
module Random_Expectation = struct
  type α mon = (α -> 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 -> 1.0/n *. k 0 +. ... +. 1.0/n *. k (n-1)

  let choose p a b = fun k -> p *. a k +. (1.0 -. p) *. b k
end
```
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 $\alpha \text{mon} = \text{state} \to (\alpha \times \text{state}) \text{outcome}$
  
  I.e. the state is discarded when we raise an exception.

- type $\alpha \text{mon} = \text{state} \to \alpha \text{outcome} \times \text{state}$
  
  I.e. the state is kept when we raise an exception.

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

$$ \text{trywith } m \ f = \lambda s. \text{match } m \ s \text{ with } \\ | (V(v), s') \to (V(v), s') \\ | (E(e), s') \to f e \left(\begin{array}{c}s \\ s'\end{array}\right) $$

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

Monad transformers

A more systematic way to build combined monads is to use monad transformers.

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 $\alpha M.\text{mon}$) into $M'$ computations (of type $\alpha M'.\text{mon}$).

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 α mon
  val ret: α -> α mon
  val bind: α mon -> (α -> β mon) -> β mon
  val run: α mon -> α
end
```

The Identity monad

The Identity monad is a trivial instance of this signature:

```
module Identity = struct
  type α mon = α
  let ret x = x
  let bind m f = f m
  let run m = m
end
```
Monad transformer for exceptions

module ExceptionTransf(M: MONAD) = struct
  type α outcome = V of α | E of exn
  type α mon = (α outcome) M.mon

  let ret x = M.ret (V x)
  let bind m f =
    M.bind m (function E e -> M.ret (E e) | V v -> f v)
  let lift x = M.bind x (fun v -> 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 -> f e | V v -> M.ret (V v))
end

Monad transformer for state

module StateTransf(M: MONAD) = struct
  type α mon = state -> (α * 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
Monad transformer for continuations

module ContTransf(M: MONAD) = struct
  type α mon = (α -> 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 α mon = state → α outcome × state, i.e. state is preserved when raising exceptions.

The other combination, StateTransf(Exception) gives α mon = state → (α × state) outcome, i.e. state is discarded when an exception is raised.
The Concurrency monad transformer

Generalizing the Continuation monad transformer, we can define concurrency (interleaving of atomic computations) as follows:

```
module Concur(M: MONAD) = struct
  type answer =
    | Seq of answer M.mon
    | Par of answer * answer
    | Stop
  type α mon = (α -> answer) -> answer
  let ret x = fun k -> k x
  let bind x f = fun k -> x (fun v -> f v k)
  let atom m = fun k -> Seq(M.bind m (fun v -> M.ret (k v)))
  let stop = fun k -> Stop
  let par m1 m2 = fun k -> Par (m1 k, m2 k)
```

If \( m : \alpha \text{ mon} \), applying \( m \) to the initial continuation \( \lambda x, \text{Stop} \) builds a tree of computations such as:

```
Seq m₁
  Par
  Seq m₂
  Seq m₄
  Stop
```

```
Seq m₃
  Par
  Seq m₅
  Stop
```

```
Seq m₆
  Stop
```

All that remains is to execute the atomic actions \( m₁, \ldots, m₆ \) in breadth-first order, simulating interleaved execution.
The Concurrency monad transformer

module Concur(M: MONAD) = struct
...
  let rec schedule acts =  
    match acts with  
    | [] -> M.ret ()  
    | Seq m :: rem ->  
        M.bind m (fun m' -> schedule (rem @ [m']))  
    | Par(a1, a2) :: rem ->  
        schedule (a1 :: a2 :: rem)  
    | Stop :: rem ->  
        schedule rem  
  let run m = M.run (schedule [m (fun _ -> Stop)])
end

Example of use

module M = Concur(Log)

let rec loop n s =  
  if n <= 0  
  then M.ret ()  
  else M.bind (M.atom (Log.log s)) (fun _ -> loop (n-1) s)

M.run (M.bind (M.atom (Log.log "start:")) (fun _ ->  
    M.par (loop 6 "a") (loop 4 "b")))

This code will log "start:ababababaaaa"