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 λ-abstractions, we have:

$$
\begin{array}{rclrclrcl}
[\![N]\!] & = & V(N) & [\![N]\!] & = & \lambda s.(N, s) & [\![N]\!] & = & \lambda k.k\ N \\
[\![x]\!] & = & V(x) & [\![x]\!] & = & \lambda s.(x, s) & [\![x]\!] & = & \lambda k.k\ x \\
[\![\lambda x.a]\!] & = & V(\lambda x.[\![a]\!]) & [\![\lambda x.a]\!] & = & \lambda s.(\lambda x.[\![a]\!], s) & [\![\lambda x.a]\!] & = & \lambda k.k\ (\lambda x.[\![a]\!])
\end{array}
$$

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:

$$
\begin{array}{rcl}
[\![\texttt{let } x = a \texttt{ in } b]\!] & = & \texttt{match } [\![a]\!] \texttt{ with } E(x) \rightarrow E(x) \mid V(x) \rightarrow [\![b]\!] \\
[\![\texttt{let } x = a \texttt{ in } b]\!] & = & \lambda s.\, \texttt{match } [\![a]\!]\ s \texttt{ with } (x, s') \rightarrow [\![b]\!]\ s' \\
[\![\texttt{let } x = a \texttt{ in } b]\!] & = & \lambda k.\, [\![a]\!]\ (\lambda x.\, [\![b]\!]\ k)
\end{array}
$$

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:

$$
\begin{array}{rcl}
[\![a\ b]\!] & = & \texttt{match } [\![a]\!] \texttt{ with} \\
& & \mid\ E(e_a) \rightarrow E(e_a) \\
& & \mid\ V(v_a) \rightarrow \\
& & \quad \texttt{match } [\![b]\!] \texttt{ with } E(e_b) \rightarrow E(e_b) \mid V(v_b) \rightarrow v_a\ v_b \\
[\![a\ b]\!] & = & \lambda s.\, \texttt{match } [\![a]\!]\ s \texttt{ with } (v_a, s') \rightarrow \\
& & \quad \texttt{match } [\![b]\!]\ s' \texttt{ with } (v_b, s'') \rightarrow v_a\ v_b\ s'' \\
[\![a\ b]\!] & = & \lambda k.\, [\![a]\!]\ (\lambda v_a.\, [\![b]\!]\ (\lambda v_b.\, v_a\ v_b\ k))
\end{array}
$$

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 α `mon` and operations `ret`, `bind` and `run`, with types:

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

The type τ `mon` is the type of computations that eventually produce a value of type τ.

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

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

`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:

$$\begin{array}{rcl} \texttt{bind}\ (\texttt{ret}\ a)\ f & \approx & f\ a \\ \texttt{bind}\ a\ (\lambda x.\ \texttt{ret}\ x) & \approx & a \\ \texttt{bind}\ (\texttt{bind}\ a\ \ (\lambda x.b))\ (\lambda y.c) & \approx & \texttt{bind}\ a\ (\lambda x.\ \texttt{bind}\ b\ (\lambda y.c)) \end{array}$$

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

Example: the Exception monad

(also called the Error monad)

```
type α mon = V of α | 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) -> f x | V(x) -> V(x)
```


Example: the State monad

```
type α mon = state → α × state

ret a = λs. (a, s)

bind m f = λs. 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 = λs. store_alloc x s
  deref r = λs. (store_read r s, s)
assign r x = λs. ((), store_write r x s)
```

Example: the Continuation monad

```
type α mon = (α → answer) → answer

ret a = λk. k a

bind m f = λk. m (λv. f v k)

run m = m (λx. x)
```

Additional operations in this monad:

```
 callcc f = λk. f k k
throw x y = λk. x y
```


Alternate presentation of a monad

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

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

The two presentations are related as follows:

$$\begin{aligned} \texttt{bind}\ a\ f &\equiv \texttt{join}(\texttt{fmap}\ f\ a) \\ \texttt{fmap}\ f\ m &\equiv \texttt{bind}\ m\ (\lambda x.\ \texttt{ret}(f\ x)) \\ \texttt{join}\ mm &\equiv \texttt{bind}\ mm\ (\lambda m.\ m) \end{aligned}$$

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

Outline

The monadic translation

Core constructs

$$\begin{array}{rcl} [\![N]\!] & = & \texttt{ret}\ \ N \\ [\![x]\!] & = & \texttt{ret}\ \ x \\ [\![\lambda x.a]\!] & = & \texttt{ret}\ \ (\lambda x.[\![a]\!]) \\ [\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] & = & \texttt{bind}\ [\![a]\!]\ (\lambda x.[\![b]\!]) \\ [\![a\ b]\!] & = & \texttt{bind}\ [\![a]\!]\ (\lambda v_a.\ \texttt{bind}\ [\![b]\!]\ (\lambda v_b.\ v_a\ v_b)) \end{array}$$

These translation rules are shared between all monads.

Effect on types: if $a : \tau$ then $[\![a]\!] : [\![\tau]\!]\ \texttt{mon}$
where $[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]\ \texttt{mon}$ and $[\![\tau]\!] = \tau$ for base types τ.

The monadic translation

Extensions

$$
\begin{array}{rcl}
[\![\mu f.\lambda x.a]\!] & = & \texttt{ret}\ \ (\mu f.\lambda x.[\![a]\!]) \\
[\![a\ op\ b]\!] & = & \texttt{bind}\ [\![a]\!]\ (\lambda v_a.\ \texttt{bind}\ [\![b]\!]\ (\lambda v_b.\ \texttt{ret}\ (v_a\ op\ v_b))) \\
[\![C(a_1,\ldots,a_n)]\!] & = & \texttt{bind}\ [\![a_1]\!]\ (\lambda v_1.\ \ldots \\
 & & \qquad \texttt{bind}\ [\![a_n]\!]\ (\lambda v_n.\ \texttt{ret}(C(v_1,\ldots,v_n))))
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{match}\ a\ \texttt{with}\ \ldots p_i \ldots]\!] & = & \texttt{bind}\ [\![a]\!]\ (\lambda v_a.\ \texttt{match}\ v_a\ \texttt{with}\ \ldots [\![p_i]\!] \ldots) \\
[\![C(x_1,\ldots,x_n) \to a]\!] & = & C(x_1,\ldots,x_n) \to [\![a]\!]
\end{array}
$$

Example of monadic translation

```
⟦1 + f  x⟧ =
    bind (ret 1) (λv1.
    bind (bind (ret f) (λv2.
          bind (ret x) (λv3. v2 v3))) (λv4.
    ret (v1 + v4)))
```

After administrative reductions using the first monadic law:

```
⟦1 + f  x⟧ =
    bind (f x) (λv. ret (1 + v))
```

Example of monadic translation

```
⟦μfact. λn. if n = 0 then 1 else n * fact(n-1)⟧ =
    ret (μfact. λn.
         if n = 0
         then ret 1
         else bind (fact(n-1)) (λv. 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:

$$\begin{array}{rcl} [\![\texttt{raise}\ a]\!] & = & \texttt{bind}\ (\texttt{ret raise})\ (\lambda v_r.\texttt{bind}\ [\![a]\!]\ (\lambda v_a.\ v_r\ \ v_a)) \\ & \stackrel{adm}{\rightarrow} & \texttt{bind}\ [\![a]\!]\ (\lambda v_a.\ \texttt{raise}\ \ v_a) \end{array}$$

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:

$$[\![\texttt{try}\ a\ \texttt{with}\ x \rightarrow b]\!] \quad = \quad \texttt{trywith}\ [\![a]\!]\ (\lambda x.[\![b]\!])$$

Syntactic properties of the monadic translation

Define the monadic translation of a value $[\![v]\!]_v$ as follows:

$$[\![N]\!]_v = N \qquad [\![\lambda x.a]\!]_v = \lambda x.[\![a]\!]$$

Lemma 1 (Translation of values)

$[\![v]\!] = \mathtt{ret}\ [\![v]\!]_v$ *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 λ-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. $(\lambda x.a)\ v \approx a[x \leftarrow v]$ (β_v reduction)
2. $\texttt{bind}\ (\texttt{ret}\ v)\ (\lambda x.b) \approx b[x \leftarrow v]$ (first monadic law)
3. $\texttt{bind}\ a\ (\lambda x.b) \approx \texttt{bind}\ a'\ (\lambda x.b)$ if $a \approx a'$ (compat. context)
4. If $a \approx \texttt{ret}\ v$, then $\texttt{run}\ a \xrightarrow{*} v$.

Correctness of the monadic translation

Theorem 3

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

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

Correctness of the monadic translation

For the `let` case:

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

The following equivalences hold:

$$\begin{array}{rcl}
[\![a]\!] & = & \texttt{bind } [\![b]\!] \ (\lambda x.[\![c]\!]) \\
\text{(ind.hyp + prop.3)} & \approx & \texttt{bind } (\texttt{ret } [\![v']\!]_v) \ (\lambda x.[\![c]\!]) \\
\text{(prop.2)} & \approx & [\![c]\!][x \leftarrow [\![v']\!]_v] \ = \ [\![c[x \leftarrow v']]\!] \\
\text{(ind.hyp.)} & \approx & \texttt{ret } [\![v]\!]_v
\end{array}$$

Correctness of the monadic translation

For the application case:

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

The following equivalences hold:

$$\begin{array}{rcl}
[\![a]\!] & = & \texttt{bind } [\![b]\!] \ (\lambda y.\texttt{bind } [\![c]\!] \ (\lambda z. \ y \ z)) \\
\text{(ind.hyp + prop.3)} & \approx & \texttt{bind } (\texttt{ret } (\lambda x.[\![d]\!])) \ (\lambda y.\texttt{bind } [\![c]\!] \ (\lambda z. \ y \ z)) \\
\text{(prop.2)} & \approx & \texttt{bind } [\![c]\!] \ (\lambda z. \ (\lambda x.[\![d]\!]) \ z)) \\
\text{(ind.hyp + prop.3)} & \approx & \texttt{bind } (\texttt{ret } [\![v']\!]_v \ (\lambda z. \ (\lambda x.[\![d]\!]) \ z)) \\
\text{(prop.2)} & \approx & (\lambda x.[\![d]\!]) \ [\![v']\!]_v \\
\text{(prop.1)} & \approx & [\![d]\!][x \leftarrow [\![v']\!]_v] \ = \ [\![d[x \leftarrow v]]\!] \\
\text{(ind.hyp.)} & \approx & \texttt{ret } [\![v]\!]_v
\end{array}$$

Correctness of the monadic translation

Theorem 4

If $a \Rightarrow v$, *then* $\texttt{run}\ [\![a]\!] \xrightarrow{*} [\![v]\!]_v$.

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.)

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 $a \xrightarrow{*} v$, then $a' \xrightarrow{*} v$.
- 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, $a_1 \xrightarrow{*} a' \xleftarrow{*} a_3$, and in the latter case, $a_1 \xrightarrow{*} a \xleftarrow{*} a_3$.
- 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

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

$$\begin{array}{ll} \multicolumn{2}{l}{\texttt{bind}\ (\texttt{ret}\ v)\ (\lambda x.b)} \\ \rightarrow & \texttt{bind}\ (V(v))\ (\lambda x.b) \\ \xrightarrow{*} & \texttt{match}\ V(v)\ \texttt{with}\ E(y) \rightarrow y \mid V(z) \rightarrow (\lambda x.b)\ z \\ \rightarrow & (\lambda x.b)\ v \rightarrow b[x \leftarrow v] \end{array}$$

3. $\texttt{bind}\ a_1\ (\lambda x.b) \approx \texttt{bind}\ a_2\ (\lambda x.b)$ if $a_1 \approx a_2$.
   Trivial since $\texttt{bind}\ [\,]\ (\lambda x.b)$ is an evaluation context.
4. If $a \approx \texttt{ret}\ v$, then $\texttt{run}\ a \xrightarrow{*} v$.
   Since $\texttt{ret}\ v \xrightarrow{*} V(v)$, we have $a \xrightarrow{*} V(v)$ and the result follows.

Application to the Continuation monad

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

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

$$\begin{array}{rll} \texttt{bind}\ (\texttt{ret}\ v)\ (\lambda x.b)\ k & \rightarrow & \texttt{bind}\ (\lambda k'.\ k'\ v)\ (\lambda x.b) \\ & \xrightarrow{*} & (\lambda k'.\ k'\ v)\ (\lambda y.\ (\lambda x.b)\ y\ k) \\ & \rightarrow & (\lambda y.\ (\lambda x.b)\ y\ k)\ v \\ & \rightarrow & (\lambda x.b)\ v\ k) \\ & \rightarrow & b[x \leftarrow v]\ k \end{array}$$

Application to the Continuation monad

1. $\mathtt{bind}\ a_1\ (\lambda x.b) \approx \mathtt{bind}\ a_2\ (\lambda x.b)$ if $a_1 \approx a_2$
   We have $\mathtt{bind}\ a_i\ (\lambda x.b)\ k \xrightarrow{*} a_i\ (\lambda v.\ (\lambda x.b)\ v\ k)$ 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 $a_i\ (\lambda v.\ (\lambda x.b)\ v\ k) \xrightarrow{*} a$ for $i = 1, 2$.
   Therefore, $\mathtt{bind}\ a_i\ (\lambda x.b)\ k \xrightarrow{*} a$ for $i = 1, 2$, and the result follows.
2. If $a \approx \mathtt{ret}\ v$, then $\mathtt{run}\ a \xrightarrow{*} v$.
   The result follows from $\mathtt{ret}\ v\ (\lambda x.x) \xrightarrow{*} v$.

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 α mon = int → α × 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
```

After monadic translation:

```
let rec insert x l =
  match l with
  | [] -> Count.ret [x]
  | h :: t ->
     Count.(tick (ret (x < h)) >>= fun b ->
            if b
            then ret (x::l)
            else insert x t >>= fun r -> ret (h::r))
```

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 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
```


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 α mon = env -> α

  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 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:

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

All permutations of a list l:

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


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 α result =
      | Success of α * char list
      | Failure
  type α mon = char list -> α result

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

  let bind (m: α mon) (f: α -> β mon): β 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
```


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` n	return a uniformly-distributed integer in $[0, n[$
`choose` p a b	evaluate either a with probability $p \in [0,1]$ or b with probability $1 - p$

In a monadic interpretation, these constructs have type

$$\begin{array}{rcl} \texttt{rand} & : & \texttt{int} \rightarrow \texttt{int mon} \\ \texttt{choose} & : & \forall\alpha.\ \texttt{float} \rightarrow \alpha\ \texttt{mon} \rightarrow \alpha\ \texttt{mon} \rightarrow \alpha\ \texttt{mon} \end{array}$$

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 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/n); ...; (n-1, 1/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 $\alpha \to$ `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/n *. k 0 +.   ...   +. 1/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:

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

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

$$\begin{array}{rcl} \texttt{trywith}\ m\ f & = & \lambda s.\ \texttt{match}\ m\ s\ \texttt{with} \\ & & \quad \mid (V(v), s') \to (V(v), s') \\ & & \quad \mid (E(e), s') \to f\ e\ \begin{pmatrix} s \\ s' \end{pmatrix} \end{array}$$

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

Composing two monads?

Given two monads

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

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

$$\texttt{type } \alpha \texttt{ mon} = \alpha \texttt{ mon1 mon2}$$

$$\texttt{let ret } (x : \alpha) : \alpha \texttt{ mon} = \texttt{ret2 } (\texttt{ret1 } x)$$

$$\begin{array}{l} \texttt{let bind } (x : \alpha \texttt{ mon}) \ (f : \alpha \to \beta \texttt{ mon}) : \beta \texttt{ mon} = \\ \quad \texttt{bind2 } x \ (\lambda y : \alpha \texttt{ mon1}. \texttt{ ret2 } (\\ \qquad \texttt{bind1 } y \ (\lambda z : \alpha. \ ???? \ (f \ z)))) \end{array}$$

Composing two monads?

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

$$\begin{array}{l} \texttt{let bind } (x : \alpha \texttt{ mon}) \ (f : \alpha \to \beta \texttt{ mon}) : \beta \texttt{ mon} = \\ \quad \texttt{bind2 } x \ (\lambda y : \alpha \texttt{ mon1}. \texttt{ ret2 } (\\ \qquad \texttt{bind1 } y \ (\lambda z : \alpha. \ ???? \ (f \ z)))) \end{array}$$

Since $f \ z : \beta \texttt{ mon1 mon2}$ and `bind1` demands something of type $\beta \texttt{ mon1}$, we need a term ???? of type $\beta \texttt{ mon1 mon2} \to \beta \texttt{ mon1}$.

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

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 α M.`mon`) into M' computations (of type α 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 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 $\texttt{mon} = \texttt{state} \to \alpha$ $\texttt{outcome} \times \texttt{state}$,
i.e. state is preserved when raising exceptions.

The other combination, `StateTransf(Exception)` gives
α $\texttt{mon} = \texttt{state} \to (\alpha \times \texttt{state})$ $\texttt{outcome}$,
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 α mon =
      | Done of α
      | Step of (α 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 α mon = Done of α | Step of (α 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.

The Concurrency monad transformer

The `ret` operation of the Concurrency monad performs zero computations:

```
let ret (x: α): α mon = Done x
```

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

```
let act (m: α M.mon): α 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)))
```

The Concurrency monad transformer

Finally, par interleaves the computations of two resumptions:

```
let rec par (m1: α mon) (m2: β mon) : (α * β) mon =
  match m1, m2 with
  | Done r1, Done r2 -> Done (r1, r2)
  | Step s1, Step s2 ->
      Step (M.bind s1 (fun m1' ->
            M.bind s2 (fun m2' -> M.ret (par m1' m2'))))
  | Done r1, Step s2 ->
      Step (M.bind s2 (fun m2' -> M.ret (par (Done r1) m2')))
  | Step s1, Done r2 ->
      Step (M.bind s1 (fun m1' -> M.ret (par m1' (Done r2))))
```


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:abababababaa"

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:

```
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 >>= fun n ->
      eval b >>= fun m ->
      ret (n + m)
```

Monadic application combinators

This can be improved by defining application combinators:

```
let mapp (f: α → β) (m: α mon) : β mon =
    bind m (fun x -> ret (f x))

let mapp2 (f: α → β → γ) (m1: α mon) (m2: β mon): γ 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: (α → β) mon) (m: α mon) : β 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 `<*>` m_1 `<*>` $\cdots$ `<*>` 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 α app of effectful computations producing a value of type α, plus two operations:

$$\begin{array}{rcl} \texttt{pure} & : & \forall\alpha,\quad \alpha \to \alpha\ \texttt{app} \\ \texttt{<*>} & : & \forall\alpha\beta,\quad (\alpha \to \beta)\ \texttt{app} \to \alpha\ \texttt{app} \to \beta\ \texttt{app} \end{array}$$

`pure` embeds values (computations without effects) into computations.

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

The laws of applicative structures

$$\begin{array}{rcl} \texttt{pure}\ (\lambda x.x)\ \texttt{<*>}\ u & \approx & u \\ \texttt{pure}\ (\lambda f.\lambda g.\lambda x.f(g\ x))\ \texttt{<*>}\ u\ \texttt{<*>}\ v\ \texttt{<*>}\ w & \approx & u\ \texttt{<*>}\ (v\ \texttt{<*>}\ w) \\ \texttt{pure}\ f\ \texttt{<*>}\ \texttt{pure}\ x & \approx & \texttt{pure}(f\ x) \\ u\ \texttt{<*>}\ \texttt{pure}\ x & \approx & \texttt{pure}(\lambda f.\ f\ x)\ \texttt{<*>}\ u \end{array}$$

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 α app = α 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.

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 α mon = V of α | 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 α mon = stringset * (env -> α)
```

(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)
```


Free variables in the Environment monad

However, `bind` cannot be defined!

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

let bind (sx,dx : α mon) (f: α -> β mon) : β mon =
  (union sx (fst (f ???)), fun e -> 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

$$\begin{array}{ll}
\texttt{type}\ \alpha\ \texttt{app1} & \texttt{type}\ \alpha\ \texttt{app2} \\
\texttt{pure1} : \alpha \to \alpha\ \texttt{app1} & \texttt{pure2} : \alpha \to \alpha\ \texttt{app2} \\
\texttt{<*>1} : (\alpha \to \beta) \to & \texttt{<*>2} : (\alpha \to \beta) \to \\
\qquad\qquad \alpha\ \texttt{app1} \to \beta\ \texttt{app1} & \qquad\qquad \alpha\ \texttt{app2} \to \beta\ \texttt{app2}
\end{array}$$

we can define

$$\texttt{type}\ \alpha\ \texttt{app} = \alpha\ \texttt{app1 app2}$$

$$\texttt{let pure}\ (x : \alpha) : \alpha\ \texttt{app} = \texttt{pure2 (pure1}\ x)$$

$$\texttt{let <*>}\ (f : \alpha \to \beta) : \alpha\ \texttt{app} \to \beta\ \texttt{app} = \texttt{<*>2 (<*>1}\ f)$$

Outline

Comonads

The categorical dual of monads (what else?)

A comonad is defined by a parameterized type α `com` and operations `proj` and `cobind`, with types:

$$\begin{aligned} \texttt{proj} &: \forall\alpha.\ \alpha\ \texttt{com} \rightarrow \alpha \\ \texttt{cobind} &: \forall\alpha,\beta.\ (\alpha\ \texttt{com} \rightarrow \beta) \rightarrow \alpha\ \texttt{com} \rightarrow \beta\ \texttt{com} \end{aligned}$$

The type τ `com` is the type of processes that produce values of type τ. (For example: a collection of τ values.)

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

`cobind` *f a*, given
– a function *f* that produces a β value from a α `com` process,
– and a α `com` process,
extends function *f* to construct a process producing β's.

Comonadic laws

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

$$\begin{aligned} \texttt{proj}(\texttt{cobind}\ k\ x) &\approx k\ x \\ \texttt{cobind}\ \texttt{proj}\ x &\approx x \\ \texttt{cobind}(k_2 \circ \texttt{cobind}\ k_1) &\approx \texttt{cobind}\ k_2 \circ \texttt{cobind}\ k_1 \end{aligned}$$

Lazy evaluation as a comonad

```
module Lazy = struct

    type α com = α status ref
    and α status =
      | Evaluated of α
      | Suspended of unit -> α

    let proj (x: α com): α =
      match !x with
      | Evaluated v -> v
      | Suspended f -> let v = f() in x := Evaluated v; v

    let cobind (f: α com -> β) (x: α com) : β com =
      ref (Suspended (fun () -> f x))

end
```


Lazy evaluation as a comonad

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

```
    let ret (x: α) : α com = ref (Evaluated x)

    let bind (x: α com) (f: α -> β com): β 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.)