Functional programming languages

Part IV: monadic transformations, monadic programming

Xavier Leroy

INRIA Rocquencourt

MPRI 2-4-2, 2006

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 and the Haskell group, 1992).

Outline

1. Introduction to monads
2. The monadic translation
   - Definition
   - Correctness
   - Application to some monads
3. Monadic programming
   - More examples of monads
   - Monad transformers

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:

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

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 whole 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:

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

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
```


Outline

1 Introduction to monads

2 The monadic translation
- Definition
- Correctness
- Application to some monads

3 Monadic programming
- More examples of monads
- Monad transformers

The monadic translation

Core constructs

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

These translation rules are shared between all monads.

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

The monadic translation

Extensions

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

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

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:

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

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:

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

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)

$[\![v]\!] = \texttt{ret}\ [\![v]\!]_v$ *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. $(\lambda x.a)\ v \approx a[x \leftarrow v]$
2. $\texttt{bind}\ (\texttt{ret}\ v)\ (\lambda x.b) \approx b[x \leftarrow v]$
3. $\texttt{bind}\ a\ (\lambda x.b) \approx \texttt{bind}\ a'\ (\lambda x.b)$ if $a \approx a'$
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 $a \Rightarrow N$, then $\texttt{run}\ [\![a]\!] \xrightarrow{*} N$.

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

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

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

Application to the Continuation monad

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

Application to the State monad

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

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

Outline

1. Introduction to monads
2. The monadic translation
   - Definition
   - Correctness
   - Application to some monads
3. Monadic programming
   - More examples of monads
   - Monad transformers

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 = m []

    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.bind (Log.log "positive") (fun _ -> n)
  else Log.bind (Log.log "negative") (fun _ -> -n)
```


The Random monad

Provides computations with a stream of pseudo-random numbers. A special case of the State monad, guaranteeing that the state of the generator is not reset during execution.

```
module Random = 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 run seed m = match m seed with (x, s') -> x

    let next_state s = s * 25173 + 1725
    let random n = fun s -> ((abs s) mod n, next_state s)
end
```

Example of use

Before monadic translation:

```
let rec randomlist n =
  if n < 0 then [] else random 10 :: randomlist (n-1)
```

After monadic translation:

```
let rec randomlist n =
  if n < 0 then Random.ret [] else
    Random.bind (Random.random 10) (fun hd ->
    Random.bind (randomlist (n-1)) (fun tl ->
    Random.ret (hd :: tl)))
```


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 (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:

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


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:

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

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

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

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 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.return (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

`ExceptionTransf` and `StateTransf` add their features "beneath" their module argument. For instance,

```
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
```

gives a type α `mon` $=$ `state` $\to$ α `outcome` $\times$ `state`, i.e. state is preserved when raising exceptions.

In contrast, `ContTransf` adds continuations "above" its module argument. For instance, `ContTransf(State)` combines continuations and state in the Scheme way: continuations transform the current state to the final state.

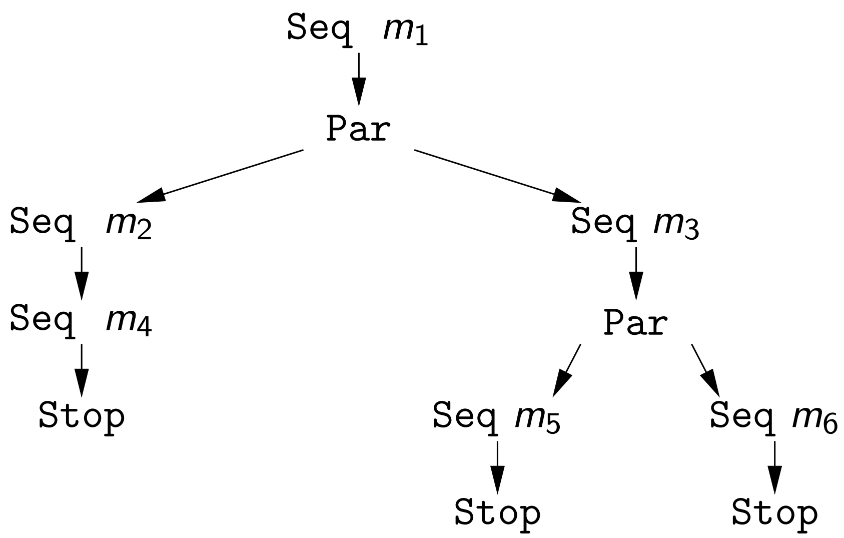
The Concurrency monad transformer

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 return x = fun k -> k x
  let bind x f = fun k -> x (fun v -> f v k)
  let atom m = fun k -> Atom(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)
```

The Concurrency monad transformer

If $m : \alpha$ `mon`, applying m to the initial continuation $\lambda x,$ `Stop` builds a tree of computations such as:



All that remains is to execute the atomic actions $m_1, \ldots, m_6$ 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:abababababaaaa`"