Monads

Outline

Monads

Exception-returning style, state-passing style, and continuation-passing style of the previous part are all special cases of *monads*

Monads are thus a technical device that factor out commonalities between many program transformations ...

Monads

Exception-returning style, state-passing style, and continuation-passing style of the previous part are all special cases of *monads*

Monads are thus a technical device that factor out commonalities between many program transformations ...

... but this is just one possible viewpoint. Besides that, they can be used

- To structure denotational semantics and make them easy to extend with new language features. (E. Moggi, 1989.)
- As a powerful programming techniques in pure functional languages, primary in Haskell. (P. Wadler, 1992).

Outline

Invent your first monad

Probably the best way to understand monads is to define one. Or better, arrive to a point where you realize that you need one (even if you do not know that it is a monad).

Many of the problems that monads try to solve are related to the issue of side effects. So we'll start with them.

Side Effects: Debugging Pure Functions

Input: We have functions `f` and `g` that both map floats to floats.

```
f,g : float -> float
```

Goal: Modify these functions to output their calls for debugging purposes

Side Effects: Debugging Pure Functions

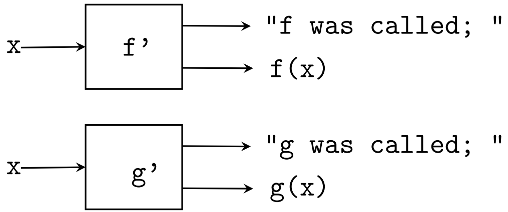
Input: We have functions `f` and `g` that both map floats to floats.

```
f,g : float -> float
```

Goal: Modify these functions to output their calls for debugging purposes

If we do not admit side effects, then the modified version `f'` and `g'` must return the output

```
f',g' : float -> float * string
```



x

f'

"f was called; "

f(x)

x

g'

"g was called; "

g(x)

We can think of these as 'debuggable' functions.

Binding

Problem: How to debug the composition of two 'debuggable' functions?

Intuition: We want the composition to have type `float -> float * string` but types no longer work!

Solution: Use concatenation for the debug messages and add some plumbing

```
let (y,s) = g' x in
let (z,t) = f' y in (z,s^t)
```

(*where* ^ *denotes string concatenation*)

Binding

Problem: How to debug the composition of two 'debuggable' functions?

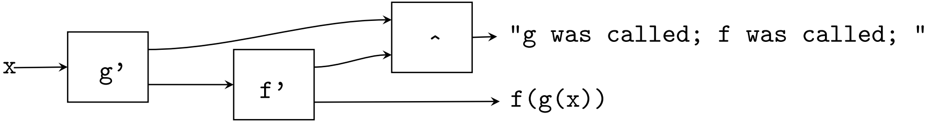
Intuition: We want the composition to have type `float -> float * string` but types no longer work!

Solution: Use concatenation for the debug messages and add some plumbing

```
let (y,s) = g' x in
let (z,t) = f' y in (z,s^t)
```

(*where ^ denotes string concatenation*)

Diagrammatically:



The `bind` function

Plumbing is ok ... once. To do it uniformly we need a higher-order function doing the plumbing for us. We need a function `bind` that upgrades `f'` so that it can be plugged in the output of `g'`. That is, we would like:

```
 bind f' : (float*string) -> (float*string)
```

which implies that

```
bind : (float ->  (float*string)) -> ( (float*string) -> (float*string))
```

`bind` must

1. apply `f'` to the correct part of `g' x` and
2. concatenate the string returned by `g'` with the string returned by `f'`.

The `bind` function

Plumbing is ok ... once. To do it uniformly we need a higher-order function doing the plumbing for us. We need a function `bind` that upgrades `f’` so that it can be plugged in the output of `g’`. That is, we would like:

```
 bind f’ : (float*string) -> (float*string)
```

which implies that

```
bind : (float ->  (float*string)) -> ( (float*string) -> (float*string))
```

`bind` must

1. apply `f’` to the correct part of `g’ x` and
2. concatenate the string returned by `g’` with the string returned by `f’`.

Exercise

Write the function bind.

The `bind` function

Plumbing is ok ... once. To do it uniformly we need a higher-order function doing the plumbing for us. We need a function `bind` that upgrades `f'` so that it can be plugged in the output of `g'`. That is, we would like:

```
 bind f' : (float*string) -> (float*string)
```

which implies that

```
bind : (float ->  (float*string)) -> ( (float*string) -> (float*string))
```

`bind` must

1. apply `f'` to the correct part of `g' x` and
2. concatenate the string returned by `g'` with the string returned by `f'`.

Exercise

Write the function bind.

```
 # let bind f' (gx,gs) = let (fx,fs) = f' gx in (fx,gs^fs)
 val bind : ('a -> 'b * string) -> 'a * string -> 'b * string = <fun>
```

The `return` function

Given two debuggable functions, `f'` and `g'`, now they can be composed by `bind`

`(bind f') .  g'` (where "`.`" is Haskell's infix composition).

Write this composition as `f'○ g'`.

We look for a "debuggable" identity function `return` such that for every debuggable function `f` one has `return ○ f = f ○ return = f`.

The `return` function

Given two debuggable functions, `f'` and `g'`, now they can be composed by `bind`

`(bind f') . g'` (where “`.`” is Haskell’s infix composition).

Write this composition as `f'` $\circ$ `g'`.

We look for a “debuggable” identity function `return` such that for every debuggable function `f` one has `return` $\circ$ `f = f` $\circ$ `return = f`.

Exercise

Define return.

The `return` function

Given two debuggable functions, `f'` and `g'`, now they can be composed by `bind`

`(bind f') . g'` (where “`.`” is Haskell’s infix composition).

Write this composition as `f'` ∘ `g'`.

We look for a “debuggable” identity function `return` such that for every debuggable function `f` one has `return` ∘ `f` = `f` ∘ `return` = `f`.

Exercise

Define return.

```
# let return x = (x,"");;
val return : 'a -> 'a * string = <fun>
```

The `return` function

Given two debuggable functions, `f’` and `g’`, now they can be composed by `bind`

`(bind f’) . g’` (where “`.`” is Haskell’s infix composition).

Write this composition as `f’` ∘ `g’`.

We look for a “debuggable” identity function `return` such that for every debuggable function `f` one has `return` ∘ `f` = `f` ∘ `return` = `f`.

Exercise

Define return.

```
# let return x = (x,"");;
val return : ’a -> ’a * string = <fun>
```

In Haskell (from now on we switch to this language):

```
Prelude> let return x = (x,"")
Prelude> :type return
return :: t -> (t, [Char])  --t is a schema variable, String = Char list
```

The `return` function

Given two debuggable functions, `f’` and `g’`, now they can be composed by `bind`

`(bind f’) . g’` (where “`.`” is Haskell’s infix composition).

Write this composition as `f’○ g’`.

We look for a “debuggable” identity function `return` such that for every debuggable function `f` one has `return ○ f = f ○ return = f`.

Exercise

Define return.

```
# let return x = (x,"");;
val return : ’a -> ’a * string = <fun>
```

In Haskell (from now on we switch to this language):

```
Prelude> let return x = (x,"")
Prelude> :type return
return :: t -> (t, [Char])  --t is a schema variable, String = Char list
```

In summary, the function `return` lifts the result of a function into the result of a “debuggable” function.

The `lift` function

The return allows us to “lift” any *function* into a debuggable one:

`let lift f = return .  f` (of type `(a -> b) -> a -> (b, [Char])`)

that is (in Ocaml) `let lift f x = (f x,"")`

The lifted version does much the same as the original function and, quite reasonably, it produces the empty string as a side effect.

The `lift` function

The return allows us to “lift” any *function* into a debuggable one:

`let lift f = return .  f`   (of type `(a -> b) -> a -> (b, [Char])`)

that is (in Ocaml) `let lift f x = (f x,"")`

The lifted version does much the same as the original function and, quite reasonably, it produces the empty string as a side effect.

Exercise

Show that `lift f` $\circ$ `lift g = lift (f.g)`

The `lift` function

The return allows us to "lift" any *function* into a debuggable one:

`let lift f = return .  f` (of type `(a -> b) -> a -> (b, [Char])`)

that is (in Ocaml) `let lift f x = (f x,"")`

The lifted version does much the same as the original function and, quite reasonably, it produces the empty string as a side effect.

Exercise

Show that `lift f` ∘ `lift g = lift (f.g)`

Summary

The functions, `bind` and `return`, allow us to compose debuggable functions in a straightforward way, and compose ordinary functions with debuggable functions in a natural way.

The `lift` function

The return allows us to “lift” any *function* into a debuggable one:

`let lift f = return .  f` (of type `(a -> b) -> a -> (b, [Char])`)

that is (in Ocaml) `let lift f x = (f x,"")`

The lifted version does much the same as the original function and, quite reasonably, it produces the empty string as a side effect.

Exercise

Show that `lift f` $\circ$ `lift g = lift (f.g)`

Summary

The functions, `bind` and `return`, allow us to compose debuggable functions in a straightforward way, and compose ordinary functions with debuggable functions in a natural way.

We just defined our first monad

The `lift` function

The return allows us to “lift” any *function* into a debuggable one:

`let lift f = return . f` (of type `(a -> b) -> a -> (b, [Char])`)

that is (in Ocaml) `let lift f x = (f x,"")`

The lifted version does much the same as the original function and, quite reasonably, it produces the empty string as a side effect.

Exercise

Show that `lift f` $\circ$ `lift g = lift (f.g)`

Summary

The functions, `bind` and `return`, allow us to compose debuggable functions in a straightforward way, and compose ordinary functions with debuggable functions in a natural way.

We just defined our first monad
Let us see more examples

Outline

A Container: Multivalued Functions

Consider `sqrt` and `cbrt` that compute the square root and cube root of a real number:

```
 sqrt,cbrt :: Float -> Float
```

Consider the complex version for these functions. They must return *lists* of results (two square roots and three cube roots)[1]

```
sqrt',cbrt' :: Complex  -> [Complex]
```

since they are *multi-valued* functions.

---

[1] `Complex` should be instead written `Complex Float`, since it is a Haskell module

A Container: Multivalued Functions

Consider `sqrt` and `cbrt` that compute the square root and cube root of a real number:

```
 sqrt,cbrt :: Float -> Float
```

Consider the complex version for these functions. They must return *lists* of results (two square roots and three cube roots)[1]

```
sqrt',cbrt' :: Complex  -> [Complex]
```

since they are *multi-valued* functions.

We can compose `sqrt` and `cbrt` to obtain the sixth root function

```
sixthrt x = sqrt (cbrt x)
```

Problem How to compose `sqrt'` and `cbrt'`?

---

[1] `Complex` should be instead written `Complex Float`, since it is a Haskell module

A Container: Multivalued Functions

Consider `sqrt` and `cbrt` that compute the square root and cube root of a real number:

```
sqrt,cbrt :: Float -> Float
```

Consider the complex version for these functions. They must return *lists* of results (two square roots and three cube roots)[1]

```
sqrt',cbrt' :: Complex  -> [Complex]
```

since they are *multi-valued* functions.

We can compose `sqrt` and `cbrt` to obtain the sixth root function

```
sixthrt x = sqrt (cbrt x)
```

Problem How to compose `sqrt'` and `cbrt'`?

Bind

We need a `bind` function that lifts `cbrt'` so that it can be applied to *all* the results of `sqrt'`

---

[1] `Complex` should be instead written `Complex Float`, since it is a Haskell module

`bind` for multivalued functions

Goal:

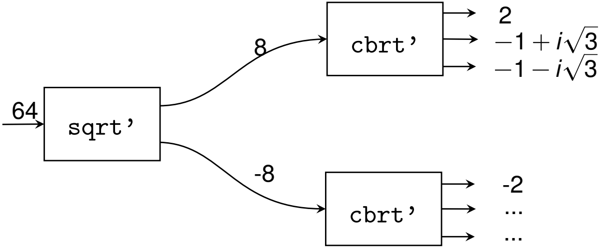
```
bind :: (Complex  -> [Complex]) -> ([Complex] -> [Complex])
```

`bind` for multivalued functions

Goal:

```
bind :: (Complex  -> [Complex]) -> ([Complex] -> [Complex])
```

Diagrammatically:

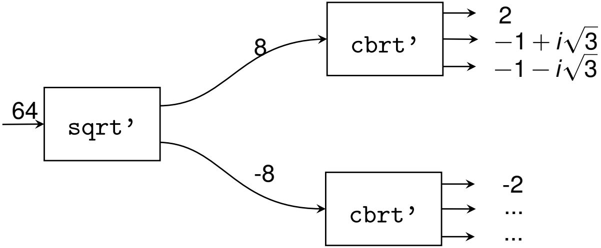


`bind` for multivalued functions

Goal:

```
bind :: (Complex  -> [Complex]) -> ([Complex] -> [Complex])
```

Diagrammatically:



Exercise

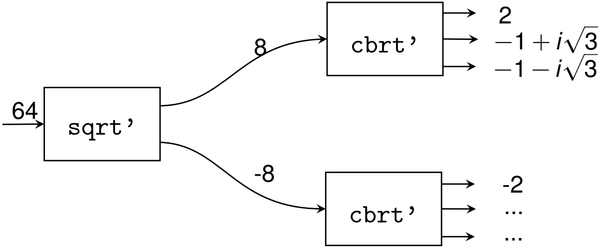
Write an implementation of `bind`

`bind` for multivalued functions

Goal:

```
bind :: (Complex  -> [Complex]) -> ([Complex] -> [Complex])
```

Diagrammatically:



Exercise

Write an implementation of `bind`

Solution:

```
bind f x = concat (map f x)
```

`return` for multivalued functions

Again we look for an identity function for multivalued functions: it takes a result of a normal function and transforms it into a result of multi-valued functions:

```
  return :: a -> [a]
```

`return` for multivalued functions

Again we look for an identity function for multivalued functions: it takes a result of a normal function and transforms it into a result of multi-valued functions:

```
return :: a -> [a]
```

Exercise

Define `return`

`return` for multivalued functions

Again we look for an identity function for multivalued functions: it takes a result of a normal function and transforms it into a result of multi-valued functions:

```
  return :: a -> [a]
```

Exercise

Define `return`

Solution:

```
return x = [x]
```

Again

$$\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$$

while `lift f = return . f` transforms an ordinary function into a multivalued one: `lift :: (a -> b) -> a -> [b]`

`return` for multivalued functions

Again we look for an identity function for multivalued functions: it takes a result of a normal function and transforms it into a result of multi-valued functions:

```
  return :: a -> [a]
```

Exercise

Define `return`

Solution:

```
return x = [x]
```

Again

$$\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$$

while `lift f = return . f` transforms an ordinary function into a multivalued one: `lift :: (a -> b) -> a -> [b]`

We just defined our second monad

`return` for multivalued functions

Again we look for an identity function for multivalued functions: it takes a result of a normal function and transforms it into a result of multi-valued functions:

```
  return :: a -> [a]
```

Exercise

Define `return`

Solution:

```
return x = [x]
```

Again

$$\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$$

while `lift f = return . f` transforms an ordinary function into a multivalued one: `lift :: (a -> b) -> a -> [b]`

We just defined our second monad
Let us see a last one and then recap

A more complex side effect: Random Numbers

The Haskell random function looks like this

```
random :: StdGen → (a,StdGen)
```

- To generate a random number you need a seed (of type `StdGen`)
- After you've generated the number you update the seed to a new value
- In a non-pure language the seed can be a global reference. In Haskell the new seed needs to be passed in and out explicitly.

A more complex side effect: Random Numbers

The Haskell random function looks like this

```
random :: StdGen → (a,StdGen)
```

- To generate a random number you need a seed (of type `StdGen`)
- After you've generated the number you update the seed to a new value
- In a non-pure language the seed can be a global reference. In Haskell the new seed needs to be passed in and out explicitly.

So a function of type `a -> b` that needs random numbers must be lifted to a "randomized" function of type `a -> StdGen -> (b,StdGen)`

Exercise

1. Write the type of the `bind` function to compose two "randomized" functions.
2. Write an implementation of `bind`

A more complex side effect: Random Numbers

Solution:

A more complex side effect: Random Numbers

Solution:

1. `bind :: (a→StdGen→(b,StdGen))`
   `→(StdGen→(a,StdGen))→(StdGen → (b,StdGen))`

A more complex side effect: Random Numbers

Solution:

1. `bind :: (a→StdGen→(b,StdGen))`
   `→(StdGen→(a,StdGen))→(StdGen → (b,StdGen))`
2. `bind f x seed =`

A more complex side effect: Random Numbers

Solution:

1.
   ```
   bind ::  (a→StdGen→(b,StdGen))
                  →(StdGen→(a,StdGen))→(StdGen → (b,StdGen))
   ```
2. `bind f x seed = let (x',seed') = x seed in f x' seed'`

A more complex side effect: Random Numbers

Solution:

1. `bind ::  (a→StdGen→(b,StdGen))`
   `→(StdGen→(a,StdGen))→(StdGen → (b,StdGen))`
2. `bind f x seed = let (x',seed') = x seed in f x' seed'`

Exercise

Define the 'identity' randomized function. This needs to be of type

`return :: a → (StdGen → (a,StdGen))`

and should leave the seed unmodified.

A more complex side effect: Random Numbers

Solution:

1. `bind :: (a→StdGen→(b,StdGen))`
   `→(StdGen→(a,StdGen))→(StdGen → (b,StdGen))`
2. `bind f x seed = let (x',seed') = x seed in f x' seed'`

Exercise

Define the 'identity' randomized function. This needs to be of type

```
return :: a → (StdGen → (a,StdGen))
```

and should leave the seed unmodified.

Solution

```
return x g = (x,g)
```

Again, `lift f = return . f` turns an ordinary function into a randomized one that leaves the seed unchanged.
While $\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$ and $\mathtt{lift}\,\mathtt{f} \circ \mathtt{lift}\,\mathtt{g} = \mathtt{lift}(\mathtt{f}.\mathtt{g})$ where $\mathtt{f} \circ \mathtt{g} = (\mathtt{bind}\,\mathtt{f}).\mathtt{g}$

Outline

29 Invent your first monad

30 More examples of monads

31 Monads and their laws

32 Program transformations and monads

33 Monads as a general programming technique

34 Monads and ML Functors

Monads

Step 1: Transform a type `a` into the type of particular *computations* on `a`.

```
-- The debuggable computations on a
type Debuggable a = (a,String)
-- The multivalued computation on a
type Multivalued a = [a]
-- The randomized computations on a
type Randomized a = StdGen -> (a,StdGen)
```

Monads

Step 1: Transform a type `a` into the type of particular *computations* on `a`.

```
-- The debuggable computations on a
type Debuggable a = (a,String)
-- The multivalued computation on a
type Multivalued a = [a]
-- The randomized computations on a
type Randomized a = StdGen -> (a,StdGen)
```

Step 2: Define the "plumbing" to lift functions on given types into functions on the "`m` computations" on these types where "`m`" is either `Debuggable`, or `Multivalued`, or `Randomized`.

```
bind :: (a -> m b) -> (m a -> m b)
return :: a -> m a
```

with $\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$ and $\mathtt{lift\ f} \circ \mathtt{lift\ g} = \mathtt{lift}\ (\mathtt{f.g})$, where '$\circ$' and `lift` are defined in terms of `return` and `bind`.

Monads

Step 1: Transform a type a into the type of particular *computations* on a.

```
-- The debuggable computations on a
type Debuggable a = (a,String)
-- The multivalued computation on a
type Multivalued a = [a]
-- The randomized computations on a
type Randomized a = StdGen -> (a,StdGen)
```

Step 2: Define the “plumbing” to lift functions on given types into functions on the “`m` computations” on these types where “`m`” is either `Debuggable`, or `Multivalued`, or `Randomized`.

```
bind :: (a -> m b) -> (m a -> m b)
return :: a -> m a
```

with $\mathtt{f} \circ \mathtt{return} = \mathtt{return} \circ \mathtt{f} = \mathtt{f}$ and $\mathtt{lift\ f} \circ \mathtt{lift\ g} = \mathtt{lift}\ (\mathtt{f.g})$, where '$\circ$' and `lift` are defined in terms of `return` and `bind`.

Monad

A *monad* is a triple formed by a type constructor `m` and two functions `bind` and `return` whose type and behavior is as described above.

Monads in Haskell

In Haskell, the `bind` function:

- it is written >>=
- it is infix
- its type is `m a -> (a -> m b) -> m b` (arguments are swapped)

Monads in Haskell

In Haskell, the `bind` function:

- it is written >>=
- it is infix
- its type is `m a -> (a -> m b) -> m b` (arguments are swapped)

This can be expressed by typeclasses:

```
class Monad m where
   -- chain computations
 (>>=)  :: m a -> ( a -> m b) -> m b
   -- inject
 return :: a -> m a
```

Monads in Haskell

In Haskell, the `bind` function:

- it is written >>=
- it is infix
- its type is `m a -> (a -> m b) -> m b` (arguments are swapped)

This can be expressed by typeclasses:

```
class Monad m where
   -- chain computations
 (>>=)  :: m a -> ( a -> m b) -> m b
   -- inject
 return :: a -> m a
```

The properties of bind and return cannot be enforced, but monadic computation demands that the following equations hold

$$\begin{aligned} \texttt{return}\ x \ \texttt{>>=}\ f &\equiv f\,x \\ m\ \texttt{>>=}\ return &\equiv m \\ m\ \texttt{>>=}\ (\lambda x.(f\,x\ \texttt{>>=}\ g)) &\equiv (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g \end{aligned}$$

Monad laws

We already saw some of these properties:

$$\begin{aligned}
\texttt{return}\ x\ \texttt{>>=}\ f &\equiv f\,x & (1)\\
m\ \texttt{>>=}\ \mathit{return} &\equiv m & (2)\\
m\ \texttt{>>=}\ (\lambda x.f\,x\ \texttt{>>=}\ g) &\equiv (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g & (3)
\end{aligned}$$

Monad laws

We already saw some of these properties:

$$\begin{aligned} \texttt{return}\ x \ \texttt{>>=}\ f &\equiv f\,x & (1)\\ m\ \texttt{>>=}\ return &\equiv m & (2)\\ m\ \texttt{>>=}\ (\lambda x.f\,x\ \texttt{>>=}\ g) &\equiv (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g & (3)\end{aligned}$$

Let us rewrite them in terms of our old `bind` function (with the different argument order we used before)

1. In (1) abstract the *x* then you have the *left identity*:

$$(\texttt{bind}\ f).\texttt{return} = f \circ \texttt{return} = f$$

Monad laws

We already saw some of these properties:

$$\begin{aligned} \texttt{return}\ x\ \texttt{>>=}\ f &\equiv f\,x && (1)\\ m\ \texttt{>>=}\ \mathit{return} &\equiv m && (2)\\ m\ \texttt{>>=}\ (\lambda x.f\,x\ \texttt{>>=}\ g) &\equiv (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g && (3)\end{aligned}$$

Let us rewrite them in terms of our old `bind` function (with the different argument order we used before)

1. In (1) abstract the x then you have the *left identity*:

$$(\texttt{bind}\ f).\texttt{return} = f \circ \texttt{return} = f$$

2. In (2) consider $m = gx$ and abstract the x then you have the *right identity*

$$(\texttt{bind}\ \texttt{return}).g = \texttt{return} \circ g = g$$

Monad laws

We already saw some of these properties:

$$\begin{aligned} \texttt{return}\ x\ \texttt{>>=}\ f &\equiv f\,x & (1)\\ m\ \texttt{>>=}\ return &\equiv m & (2)\\ m\ \texttt{>>=}\ (\lambda x.f\,x\ \texttt{>>=}\ g) &\equiv (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g & (3)\end{aligned}$$

Let us rewrite them in terms of our old `bind` function (with the different argument order we used before)

1. In (1) abstract the x then you have the *left identity*:

   $$(\texttt{bind}\ f).\texttt{return} = f \circ \texttt{return} = f$$

2. In (2) consider $m = gx$ and abstract the x then you have the *right identity*

   $$(\texttt{bind}\ \texttt{return}).g = \texttt{return} \circ g = g$$

3. Law (3) express *associativity* (exercise: prove it)

   $$h \circ (f \circ g) = (h \circ f) \circ g$$

Writer, List and State Monads

The monads we showed are special cases of Writer, List, and State monads. Let us see their (simplified) versions

```
-- The Writer Monad
data Writer a = Writer (a, [Char])

instance Monad Writer where
   return x              = Writer (x,[])
   Writer (x,l) >>= f    = let Writer (x',l') = f x in Writer (x', l++l')


-- The List monad ([] data type is predefined)
instance Monad [] where
    return x             = [x]
    m >>= f              = concat (map f m)


-- The State Monad
data State s a = State (s -> (a,s))

instance Monad (State s) where
    return a             = State (λs -> (a,s))                -- \s -> (a,s)
    (State g) >>= f      = State (λs -> let (v,s') = g s in
                                        let State h = f v in h s')
```

Back to program transformations

QUESTION

Haven’t you already seen the state monad?

Back to program transformations

QUESTION

Haven't you already seen the state monad?

Let us strip out the type constructor part:

```
return a      = λs -> (a,s)
a >>= f       = λs -> let (v,s') = a s in (f v) s'
```

Back to program transformations

QUESTION

Haven't you already seen the state monad?

Let us strip out the type constructor part:

```
return a     = λs -> (a,s)
a >>= f      = λs -> let (v,s') = a s in (f v) s'
```

It recalls somehow the transformation for the state passing style:

$$\begin{array}{rcl}
[\![N]\!] & = & \lambda s.(N,s) \\
[\![x]\!] & = & \lambda s.(x,s) \\
[\![\lambda x.a]\!] & = & \lambda s.(\lambda x.[\![a]\!],s) \\
[\![\texttt{let } x = a \texttt{ in } b]\!] & = & \lambda s.\texttt{match } [\![a]\!]s \texttt{ with } (x,s') \to [\![b]\!]s' \\
[\![ab]\!] & = & \lambda s.\texttt{match } [\![a]\!]s \texttt{ with } (x_a,s') \to \\
 & & \quad \texttt{match } [\![b]\!]s' \texttt{ with } (x_b,s'') \to x_a\,x_b\,s''
\end{array}$$

Back to program transformations

QUESTION

Haven't you already seen the state monad?

Let us strip out the type constructor part:

```
return a     = λs -> (a,s)
a >>= f      = λs -> let (v,s') = a s in (f v) s'
```

It recalls somehow the transformation for the state passing style:

$$
\begin{array}{rcl}
[\![N]\!] & = & \lambda s.(N,s) \\
[\![x]\!] & = & \lambda s.(x,s) \\
[\![\lambda x.a]\!] & = & \lambda s.(\lambda x.[\![a]\!],s) \\
[\![\texttt{let}\ x=a\ \texttt{in}\ b]\!] & = & \lambda s.\texttt{match}\ [\![a]\!]s\ \texttt{with}\ (x,s')\rightarrow[\![b]\!]s' \\
[\![ab]\!] & = & \lambda s.\texttt{match}\ [\![a]\!]s\ \texttt{with}\ (x_a,s')\rightarrow \\
 & & \qquad \texttt{match}\ [\![b]\!]s'\ \texttt{with}\ (x_b,s'')\rightarrow x_a\,x_b\,s''
\end{array}
$$

Exactly the same transformation but with different constructions

Outline

29 Invent your first monad

30 More examples of monads

31 Monads and their laws

32 Program transformations and monads

33 Monads as a general programming technique

34 Monads and ML Functors

Commonalities of program transformations

Let us temporary abandon Haskell and return to pseudo-OCaml syntax

Consider the conversions to exception-returning style, state-passing style, and continuation-passing style. For constants, variables and λ-abstractions (ie., *values*), we have:

Pure	**Exceptions**	**State**	**Continuations**
$[\![N]\!]$	$= \quad Val(N)$	$= \quad \lambda s.(N, s)$	$= \quad \lambda k.kN$
$[\![x]\!]$	$= \quad Val(x)$	$= \quad \lambda s.(x, s)$	$= \quad \lambda k.kx$
$[\![\lambda x.a]\!]$	$= \quad Val(\lambda x.[\![a]\!])$	$= \quad \lambda s.(\lambda x.[\![a]\!], s)$	$= \quad \lambda k.k(\lambda x.[\![a]\!])$

In all three cases we **`return`** the values N, x, or $\lambda x.[\![a]\!]$ wrapped in some appropriate context.

Commonalities of program transformations

For `let` bindings we have

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

In all three cases we extract the value resulting from the computation $[\![a]\!]$, we **bind** it to the variable x and proceed with the computation $[\![b]\!]$.

Commonalities of program transformations

For applications we have

$$
\begin{array}{rcl}
[\![a\,b]\!] & = & \texttt{match } [\![a]\!] \texttt{ with} \\
& & \mid Exn(x_a) \to Exn(x_a) \\
& & \mid Val(x_a) \to \texttt{match } [\![b]\!] \texttt{ with} \\
& & \qquad\qquad \mid Exn(y_b) \to Exn(y_b) \\
& & \qquad\qquad \mid Val(y_b) \to x_a\, y_b \\
[\![a\,b]\!] & = & \lambda s.\texttt{match } [\![a]\!]s \texttt{ with } (x_a, s') \to \\
& & \quad \texttt{match } [\![b]\!]s' \texttt{ with } (y_b, s'') \to x_a\, y_b\, s'' \\
[\![a\ b]\!] & = & \lambda k.[\![a]\!](\lambda x_a.[\![b]\!](\lambda y_b.x_a\, y_b\, k))
\end{array}
$$

We **`bind`** the value of $[\![a]\!]$ to the variable x_a, then **`bind`** the value of $[\![b]\!]$ to the variable y_b, then perform the application $x_a y_b$, and rewrap the result as needed.

Commonalities of program transformations

For types notice that if $a : \tau$ then $[\![a]\!] : [\![\tau]\!]\ \texttt{mon}$
where
- $[\![\tau_1 \to \tau_2]\!] = \tau_1 \to [\![\tau_2]\!]\ \texttt{mon}$
- $[\![B]\!] = B$ for bases types B.

For exceptions:

```
type α mon = Val of α | Exn of exn
```

For states:

```
type α mon  = state → α × state
```

For continuations:

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

Monadic translation

The previous three translations are instances of the following translation

$$\begin{array}{rcl}
[\![N]\!] & = & \texttt{return}\ N \\
[\![x]\!] & = & \texttt{return}\ x \\
[\![\lambda x.a]\!] & = & \texttt{return}\ (\lambda x.[\![a]\!]) \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] & = & [\![a]\!]\ \texttt{>>=}\ (\lambda x.[\![b]\!]) \\
[\![a\,b]\!] & = & [\![a]\!]\ \texttt{>>=}\ (\lambda x_a.[\![b]\!]\ \texttt{>>=}\ (\lambda y_b.x_a y_b))
\end{array}$$

just the monad changes, that is, the definitions of `bind` and `return`).

Exception monad

So the previous translation coincides with our exception returning transformation for the following definitions of bind and return:

```
type α mon = Val of α | Exn of exn

return a   = Val(a)

m >>= f    = match m with Exn(x) -> Exn(x) | Val(x) -> f x
```

Exception monad

So the previous translation coincides with our exception returning transformation for the following definitions of bind and return:

```
type α mon = Val of α | Exn of exn

return a   = Val(a)

m >>= f    = match m with Exn(x) -> Exn(x) | Val(x) -> f x
```

bind encapsulates the propagation of exceptions in compound expressions such as the application *ab* or let bindings. As usual we have:

```
return : α → α mon

(>>=)  : α mon → (α → β mon) → β mon
```

Exception monad

So the previous translation coincides with our exception returning transformation for the following definitions of bind and return:

```
type α mon = Val of α | Exn of exn

return a   = Val(a)

m >>= f    = match m with Exn(x) -> Exn(x) | Val(x) -> f x
```

bind encapsulates the propagation of exceptions in compound expressions such as the application *ab* or let bindings. As usual we have:

```
return : α → α mon

(>>=)  : α mon → (α → β mon) → β mon
```

Additional operations in this monad:

```
raise x = Exn(x)

trywith m f = match m with Exn(x) -> f x | Val(x) -> Val(x)
```

The State monad

To have the state-passing transformation we use instead the following definitions for return and bind:

```
type α mon = state → α × state

return a = λs. (a, s)

m >>= f = λs. match m s with (x, s’) -> f x s’
```

bind encapsulates the threading of the state in compound expressions.

The State monad

To have the state-passing transformation we use instead the following definitions for return and bind:

```
type α mon = state → α × state

return a = λs. (a, s)

m >>= f = λs. match m s with (x, s’) -> f x s’
```

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
```

The Continuation monad

Finally the following monad instance yields the continuation-passing transformation:

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

return a = λk. k a

m >>= f = λk. m (λv. f v k)
```

Additional operations in this monad:

```
callcc f = λk. f k k

throw x y = λk. x y
```

More on monadic translation

We can extend the monadic translation to more constructions of the language.

$$
\begin{array}{rcl}
[\![\mu f.\lambda x.a]\!] & = & \mathtt{return}(\mu f.\lambda x.[\![a]\!]) \\
[\![a\ \mathbf{op}\ b]\!] & = & [\![a]\!]\ \mathtt{>>=}\ (\lambda x_a.[\![b]\!]\ \mathtt{>>=}\ (\lambda y_b.\mathtt{return}(x_a\ \mathbf{op}\ y_b))) \\
[\![C(a_1,\ldots,a_n)]\!] & = & [\![a_1]\!]\ \mathtt{>>=}\ (\lambda x_1.\ldots[\![a_n]\!]\ \mathtt{>>=}\ (\lambda x_n.\mathtt{return}(C(x_1,\ldots,x_n))) \\
[\![\mathtt{match}\ a\ \mathtt{with}\ ..p..]\!] & = & [\![a]\!]\ \mathtt{>>=}\ (\lambda x_a.\mathtt{match}\ x_a\ \mathtt{with}\ ..[\![p]\!]...) \\
 & & \qquad \text{where } [\![C(x_1,\ldots,x_n) \to a]\!] = C(x_1,\ldots,x_n) \to [\![a]\!]
\end{array}
$$

All these are parametric in the definition of bind and return.

Correctness of the monadic translation

The fundamental property of the monadic translation is that it does not alter the semantics of the computation it encodes. It just adds to the computation some effects.

Theorem

If $a \Rightarrow v$ *, then* $[\![a]\!] \equiv \texttt{return}\ v'$

$$\text{where } v' = \begin{cases} N & \text{if } v = N \\ \lambda x.[\![a]\!] & \text{if } v = \lambda x.a \end{cases}$$

Examples of monadic translation

```
⟦ 1 + f x ⟧  =
      (return 1) >>= (λx_1.
      ((return f) >>= (λx_2.
                  (return x) >>= (λx_3. x_2 x_3))) >>=( λx_4.
      return (x_1 + x_4)))
```

After administrative reductions using the first monadic law:

(`return x >>= f` is equivalent to `f x`)

```
⟦ 1 + f x ⟧  =
      (f x) >>= (λx_4. return (1 + x_4))
```

Examples of monadic translation

```
⟦ 1 + f x ⟧  =
      (return 1) >>= (λx_1.
      ((return f) >>= (λx_2.
                  (return x) >>= (λx_3. x_2 x_3))) >>=( λx_4.
      return (x_1 + x_4)))
```

After administrative reductions using the first monadic law:

(`return x >>= f` is equivalent to `f x`)

```
⟦ 1 + f x ⟧  =
      (f x) >>= (λx_4. return (1 + x_4))
```

A second example

```
⟦ μfact. λn. if n = 0 then 1 else n * fact(n-1) ⟧  =
      return (μfact. λn.
                if n = 0
                then return 1
                else (fact(n-1)) >>= (λv. return (n * v))
             )
```

Summary

What we have done:

1. Take a program that performs some computation
2. Apply the monadic transformation to it. This yields a new program that uses `return` and >>= in it.
3. Choose a monad (that is, choose a definition for `return` and `>>=`) and the new programs embeds the computation in the corresponding monad (side-effects, exceptions, etc.)
4. You can now add in the program the operations specific to the chosen monad: although it includes effects the program is still *pure*.

Outline

Monads as a general programming technique

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

- the proper algorithms, and
- the “plumbing” needed by *computation* of these algorithms to produce effects (state passing, exception handling, non-deterministic choice, etc).

Monads as a general programming technique

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

- the proper algorithms, and
- the “plumbing” needed by *computation* of these algorithms to produce effects (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 parametrized over a monad and reused with several monads.
- Monads themselves can be built in an incremental manner.

Monads as a general programming technique

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

- the proper algorithms, and
- the “plumbing” needed by *computation* of these algorithms to produce effects (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 parametrized over a monad and reused with several monads.
- Monads themselves can be built in an incremental manner.

Back to Haskell

Let us put all this at work by writing in Haskell the canonical, efficient interpreter that ended our refresher course on operational semantics.

The canonical, efficient interpreter in OCaml (reminder)

```
# type term = Const of int | Var of int | Abs of term
            | App of term * term | Plus of term * term
   and value = Vint of int | Vclos of term * environment
   and environment = value list                      (* use Vec instead *)

# exception Error

# let rec eval e a =                        (* : environment -> term -> value *)
    match a with
    | Const n -> Vint n
    | Var n -> List.nth e n
    | Abs a -> Vclos(Abs a, e)
    | App(a, b) -> ( match eval e a with
        | Vclos(Abs c, e’) ->
            let v = eval e b in
            eval (v :: e’) c
        | _ -> raise Error)
    | Plus(a,b) ->  match (eval e a, eval e b) with
        | (Vint n, Vint m) -> Vint (n+m)
        |  _ -> raise Error

# eval [] (Plus(Const(5),(App(Abs(Var 0),Const(2)))));;(* 5+((λx.x)2)→7 *)
- : value = Vint 7
```

Note:a `Plus` operator added and used `Abs` instead of `Lam`

The canonical, efficient interpreter in Haskell

```
data Exp   = Const Integer                            -- expressions
           | Var Integer
           | Plus Exp Exp
           | Abs Exp
           | App Exp Exp
data Value = Vint Integer                             -- values
           | Vclos Env Exp

type Env   = [Value]                                  -- list of values

eval0 :: Env -> Exp -> Value
eval0 env (Const i )     = Vint i
eval0 env (Var n)        = env !! n                    -- n-th element
eval0 env (Plus e1 e2 ) = let Vint i1 = eval0 env e1
                              Vint i2 = eval0 env e2   -- let syntax
                           in Vint (i1 + i2 )
eval0 env (Abs e)        = Vclos env e
eval0 env (App e1 e2 )  = let Vclos env0 body  = eval0 env e1
                              val = eval0 env e2
                           in eval0 (val : env0) body
```

The canonical, efficient interpreter in Haskell

```
data Exp   = Const Integer                            -- expressions
           | Var Integer
           | Plus Exp Exp
           | Abs Exp
           | App Exp Exp
data Value = Vint Integer                             -- values
           | Vclos Env Exp

type Env   = [Value]                                  -- list of values

eval0 :: Env -> Exp -> Value
eval0 env (Const i )    = Vint i
eval0 env (Var n)       = env !! n                     -- n-th element
eval0 env (Plus e1 e2 ) = let Vint i1 = eval0 env e1
                              Vint i2 = eval0 env e2   -- let syntax
                           in Vint (i1 + i2 )
eval0 env (Abs e)       = Vclos env e
eval0 env (App e1 e2 )  = let Vclos env0 body  = eval0 env e1
                              val = eval0 env e2
                           in eval0 (val : env0) body
```

No exceptions: pattern matching may fail.

```
*Main> eval0 [] (App (Const 3) (Const 4))
*** Irrefutable pattern failed for pattern Main.Vclos env body
```

Haskell "`do`" Notation

Haskell has a very handy notation for monads

In a `do` block you can macro expand every intermediate line of the form

pattern `<-` *expression*	into	*expression* `>>= \` *pattern* `->`

and every intermediate line of the form

expression	into	*expression* `>>= \ _ ->`

Haskell “`do`” Notation

Haskell has a very handy notation for monads

In a `do` block you can macro expand every intermediate line of the form

pattern `<-` *expression* into *expression* `>>=` `\` *pattern* `->`

and every intermediate line of the form

expression into *expression* `>>=` `\ _ ->`

This allows us to simplify the monadic translation for expressions which in Haskell syntax is defined as

$$
\begin{aligned}
\llbracket N \rrbracket &= \texttt{return}\ N \\
\llbracket x \rrbracket &= \texttt{return}\ x \\
\llbracket \lambda x.a \rrbracket &= \texttt{return}\ (\backslash x\texttt{->}\llbracket a \rrbracket) \\
\llbracket \texttt{let}\ x = a\ \texttt{in}\ b \rrbracket &= \llbracket a \rrbracket\ \texttt{>>=}\ (\backslash x\texttt{->}\llbracket b \rrbracket) \\
\llbracket a\, b \rrbracket &= \llbracket a \rrbracket\ \texttt{>>=}\ (\backslash x_a\texttt{->}\llbracket b \rrbracket\ \texttt{>>=}\ (\backslash y_b\texttt{->}x_a y_b))
\end{aligned}
$$

By using the `do` notation the last two cases become far simpler to understand

Monadic transformation in Haskell

$$
\begin{array}{rcl}
[\![N]\!] & = & \texttt{return } N \\
[\![x]\!] & = & \texttt{return } x \\
[\![\lambda x.a]\!] & = & \texttt{return } (\texttt{\\}x\texttt{->}[\![a]\!]) \\
[\![\texttt{let } x = a \texttt{ in } b]\!] & = & \texttt{do } \; x \texttt{ <- } [\![a]\!] \\
 & & \qquad [\![b]\!] \\
[\![a\,b]\!] & = & \texttt{do } \; x_a \texttt{ <- } [\![a]\!] \\
 & & \qquad y_b \texttt{ <- } [\![b]\!] \\
 & & \qquad x_a\,y_b
\end{array}
$$

The translation shows that `do` is the monadic version of `let`.

Monadic transformation in Haskell

$$
\begin{array}{rcl}
[\![N]\!] & = & \texttt{return}\ N \\
[\![x]\!] & = & \texttt{return}\ x \\
[\![\lambda x.a]\!] & = & \texttt{return}\ (\texttt{\\}x\texttt{->}[\![a]\!]) \\
[\![\texttt{let}\ x = a\ \texttt{in}\ b]\!] & = & \texttt{do}\ \ x\ \texttt{<-}\ [\![a]\!] \\
 & & \qquad [\![b]\!] \\
[\![a\,b]\!] & = & \texttt{do}\ \ x_a\ \texttt{<-}\ [\![a]\!] \\
 & & \qquad y_b\ \texttt{<-}\ [\![b]\!] \\
 & & \qquad x_a\,y_b
\end{array}
$$

The translation shows that `do` is the monadic version of `let`.

Monad at work

Let us apply the transformation to our canonical efficient interpreter

The canonical, efficient interpreter in monadic form

```
newtype Identity a = MkId a

instance Monad Identity where
    return a       = MkId a           -- i.e. return = id
    (MkId x) >>= f = f x              -- i.e. x >>= f = f x

eval1 :: Env -> Exp -> Identity Value
eval1 env (Const i )     = return (Vint i)
eval1 env (Var n)        = return (env !! n)
eval1 env (Plus e1 e2 ) = do Vint i1  <- eval1 env e1
                             Vint i2  <- eval1 env e2
                             return (Vint (i1 + i2 ))
eval1 env (Abs e)        = return (Vclos env e)
eval1 env (App e1 e2 )  = do Vclos env0 body  <- eval1 env e1
                             val  <- eval1 env e2
                             eval1 (val : env0 ) body
```

The canonical, efficient interpreter in monadic form

```
newtype Identity a = MkId a

instance Monad Identity where
    return a         = MkId a               -- i.e. return = id
    (MkId x) >>= f = f x                   -- i.e. x >>= f = f x

eval1 :: Env -> Exp -> Identity Value
eval1 env (Const i )     = return (Vint i)
eval1 env (Var n)        = return (env !! n)
eval1 env (Plus e1 e2 ) = do Vint i1  <- eval1 env e1
                             Vint i2  <- eval1 env e2
                             return (Vint (i1 + i2 ))
eval1 env (Abs e)        = return (Vclos env e)
eval1 env (App e1 e2 )  = do Vclos env0 body  <- eval1 env e1
                             val  <- eval1 env e2
                             eval1 (val : env0 ) body
```

We just replaced “`do`” for “`let`”, replaced “`<-`” for “`=`”, and put “`return`” in front of every value returned.

The canonical, efficient interpreter in monadic form

```
newtype Identity a = MkId a

instance Monad Identity where
    return a       = MkId a          -- i.e. return = id
    (MkId x) >>= f = f x             -- i.e. x >>= f = f x

eval1 :: Env -> Exp -> Identity Value
eval1 env (Const i )    = return (Vint i)
eval1 env (Var n)       = return (env !! n)
eval1 env (Plus e1 e2 ) = do Vint i1  <- eval1 env e1
                             Vint i2  <- eval1 env e2
                             return (Vint (i1 + i2 ))
eval1 env (Abs e)       = return (Vclos env e)
eval1 env (App e1 e2 )  = do Vclos env0 body  <- eval1 env e1
                             val  <- eval1 env e2
                             eval1 (val : env0 ) body
```

We just replaced "`do`" for "`let`", replaced "`<-`" for "`=`", and put "`return`" in front of every value returned. Let us try to execute $(\lambda x.(x+1))4$

```
*Main> let MkId x = (eval1 [] (App(Abs(Plus(Var 0)(Const 1)))(Const 4)))
       in x
Vint 5
```

Although we wrote `eval1` for the Identity monad, the type of eval1 could be generalized to

```
eval1 :: Monad m => Env -> Exp -> m Value,
```

because we do not use any monadic operations other than `return` and `>>=` (hidden in the `do` notation): no `raise`, `assign`, `trywith`, ...
Recall that the type

```
Monad m => Env -> Exp -> m Value,
```

reads “for every type (constructor) `m` that is an instance of the type class `Monad`, the function has type `Env -> Exp -> m Value`”.

Although we wrote `eval1` for the Identity monad, the type of eval1 could be generalized to

```
eval1 :: Monad m => Env -> Exp -> m Value,
```

because we do not use any monadic operations other than `return` and `>>=` (hidden in the `do` notation): no `raise`, `assign`, `trywith`, ...
Recall that the type

```
Monad m => Env -> Exp -> m Value,
```

reads “for every type (constructor) `m` that is an instance of the type class `Monad`, the function has type `Env -> Exp -> m Value`”.
In our first definition of `eval1` we explicitly instantiated `m` into the Identity monad, but we can let the system instantiate it. For instance, if we give `eval` the generalized type above, then we do not need to extract the value encapsulated in the effect:

```
*Main>  (eval1 [] (App(Abs(Plus(Var 0)(Const 1)))(Const 4)))
Vint 5
```

The `ghci` prompt has run the expression in (ie, instantiated `m` by) the IO monad, because internally the interpreter uses the print function, which lives in just this monad.

Instantiating `eval` with the `Exception` monad

We decide to instantiate `m` in `eval` with the following monad:

```
data Exception e a = Val a | Exn e

instance Monad (Exception e) where
    return x    = Val x
    m >>= f     = case m of
                     Exn x -> Exn x
                     Val x -> f x

raise :: e -> Exception e a
raise x = Exn x

trywith :: Exception e a -> (e -> Exception e a) -> Exception e a
trywith m f = case m of
                 Exn x -> f x
                 Val x -> Val x
```

Note: Haskell provides an `Error` monad for exceptions. Not dealt with here.

Instantiating `eval` with the `Exception` monad

We can do dull instantiation:

```
eval1 :: Env -> Exp -> Exception e Value
eval1 env (Const i )     = return (Vint i)
eval1 env (Var n)        = return (env !! n)
eval1 env (Plus e1 e2 ) = do Vint i1  <- eval1 env e1
                             Vint i2  <- eval1 env e2
                             return (Vint (i1 + i2))
eval1 env (Abs e)        = return (Vclos env e)
eval1 env (App e1 e2 )  = do Vclos env0 body  <- eval1 env e1
                             val <- eval1 env e2
                             eval1 (val : env0) body
```

Instantiating `eval` with the `Exception` monad

We can do dull instantiation:

```
eval1 :: Env -> Exp -> Exception e Value
eval1 env (Const i )    = return (Vint i)
eval1 env (Var n)       = return (env !! n)
eval1 env (Plus e1 e2 ) = do Vint i1  <- eval1 env e1
                             Vint i2  <- eval1 env e2
                             return (Vint (i1 + i2))
eval1 env (Abs e)       = return (Vclos env e)
eval1 env (App e1 e2 )  = do Vclos env0 body  <- eval1 env e1
                             val <- eval1 env e2
                             eval1 (val : env0) body
```

Not interesting since all we obtained is to encapsulate the result into a `Val` constructor.

Instantiating `eval` with the `Exception` monad

We can do dull instantiation:

```
eval1 :: Env -> Exp -> Exception e Value
eval1 env (Const i )     = return (Vint i)
eval1 env (Var n)        = return (env !! n)
eval1 env (Plus e1 e2 ) = do Vint i1  <- eval1 env e1
                             Vint i2  <- eval1 env e2
                             return (Vint (i1 + i2))
eval1 env (Abs e)        = return (Vclos env e)
eval1 env (App e1 e2 )  = do Vclos env0 body  <- eval1 env e1
                             val <- eval1 env e2
                             eval1 (val : env0) body
```

Not interesting since all we obtained is to encapsulate the result into a `Val` constructor.

The smart way

Use the exception monad to do as the OCaml implementation and raise an error when the applications are not well-typed

Instantiating `eval` with the `Exception` monad

New interpreter with exceptions:

```
eval2 :: Env -> Exp -> Exception String Value   -- exceptions as strings
eval2 env (Const i )    = return (Vint i)
eval2 env (Var n)       = return (env !! n)
eval2 env (Plus e1 e2 ) = do x1 <- eval2 env e1
                             x2 <- eval2 env e2
                             case (x1 , x2) of
                                (Vint i1, Vint i2)
                                  -> return (Vint (i1 + i2))
                                _ -> raise "type error in addition"
eval2 env (Abs e)       = return (Vclos env e)
eval2 env (App e1 e2 )  = do fun <- eval2 env e1
                             val <- eval2 env e2
                             case fun of
                                Vclos env0 body
                                  -> eval2 (val : env0) body
                                _ -> raise "type error in application"
```

Instantiating `eval` with the `Exception` monad

New interpreter with exceptions:

```
eval2 :: Env -> Exp -> Exception String Value   -- exceptions as strings
eval2 env (Const i )     = return (Vint i)
eval2 env (Var n)        = return (env !! n)
eval2 env (Plus e1 e2 ) = do x1 <- eval2 env e1
                             x2 <- eval2 env e2
                             case (x1 , x2) of
                                (Vint i1, Vint i2)
                                  -> return (Vint (i1 + i2))
                                _ -> raise "type error in addition"
eval2 env (Abs e)        = return (Vclos env e)
eval2 env (App e1 e2 )  = do fun <- eval2 env e1
                             val <- eval2 env e2
                             case fun of
                                Vclos env0 body
                                  -> eval2 (val : env0) body
                                _ -> raise "type error in application"
```

And we see that the exception is correctly raised

```
*Main>  let Val x = ( eval2 [] (App (Abs (Var 0)) (Const 3)) ) in x
Vint 3
*Main>  let Exn x = ( eval2 [] (App (Const 2) (Const 3)) ) in x
"type error in application"
```

Instantiating `eval` with the `Exception` monad

Advantages:

- The function `eval2` is *pure*!
- Module few syntactic differences the code is really the same as code that would be written in an impure language (*cf*. the corresponding OCaml code)
- All "plumbing" necessary to preserve purity is defined separately (eg, in the `Exception` monad and its extra functions)
- In most cases the programmer does not even need to define "plumbing" since monads provided by standard Haskell libraries are largely sufficient.

Instantiating `eval` with the `Exception` monad

Advantages:

- The function `eval2` is *pure*!
- Module few syntactic differences the code is really the same as code that would be written in an impure language (*cf*. the corresponding OCaml code)
- All “plumbing” necessary to preserve purity is defined separately (eg, in the `Exception` monad and its extra functions)
- In most cases the programmer does not even need to define “plumbing” since monads provided by standard Haskell libraries are largely sufficient.

A second try

Let us instantiate the type `Monad m => Env -> Exp -> m Value` with a different monad `m`. For our next example we choose the `State` monad.

Instantiating `eval` with the `State` monad

Goal: Add profiling capabilities by recording the number of evaluation steps.

```
-- The State Monad
data State s a = MkSt (s -> (a,s))

instance Monad (State s) where
    return a        = MkSt (\s -> (a,s))
    (MkSt g) >>= f  = MkSt (\s -> let (v,s') = g s
                                      MkSt h = f v
                                   in h s')

get :: State s s
get = MkSt (\s -> (s,s))

put :: s -> State s ()
put s = MkSt (\_ -> ((),s))
```

To count evaluation steps we use an Integer number as state (ie, we use the `State Integer` monad). The operation `tick`, retrieves the hidden state from the computation, increases it and stores it back

```
tick :: State Integer ()
tick = do st <- get
          put (st + 1)
```

Instantiating `eval` with the `State` monad

```
eval3 :: Env -> Exp -> State Integer Value
eval3 env (Const i )     = do tick
                              return (Vint i)
eval3 env (Var n)        = do tick
                              return (env !! n)
eval3 env (Plus e1 e2 ) = do tick
                              x1 <- eval3 env e1
                              x2 <- eval3 env e2
                              case (x1 , x2) of
                                 (Vint i1, Vint i2)
                                    -> return (Vint (i1 + i2 ))
eval3 env (Abs e)        = do tick
                              return (Vclos env e)
eval3 env (App e1 e2 )  = do tick
                              fun <- eval3 env e1
                              val <- eval3 env e2
                              case fun of
                                 Vclos env0 body
                                    -> eval3 (val : env0 ) body
```

The evaluation of $(\lambda x.x)3$ takes 4 steps of reduction. This is shown by giving 0 as initial value of the state:

```
*Main>  let MkSt s = eval3 [] (App (Abs (Var 0)) (Const 3)) in s 0
(Vint 3,4)
```

Combining monads the hard way

What if we want *both* exceptions and state in our interpreter?

- Merging the code of `eval2` and `eval3` is straightforward: just add the code of `eval2` that raises the type-error exceptions at the end of the `Plus` and `App` cases in the definition of `eval3`.
- The problem is how to define the monad that supports both effects.

Combining monads the hard way

What if we want *both* exceptions and state in our interpreter?

- Merging the code of `eval2` and `eval3` is straightforward: just add the code of `eval2` that raises the type-error exceptions at the end of the `Plus` and `App` cases in the definition of `eval3`.
- The problem is how to define the monad that supports both effects.

We can *write from scratch* the monad `m` that supports both effects.

```
eval4 :: Monad m => Env -> Exp -> m Value
```

Where the monad `m` above is one of the following two cases:

1. Use `StateOfException s e` for `m`: (with `s=Integer` and `e=[Char]`)

   ```
   data StateOfException s e a = State (s -> Exception e (s,a))
   ```

   the computation can either return a new pair state, value or generate an error (ie, when an exception is raised the state is discarded)
2. Use `ExceptionOfState s e` for `m`: (with `s=Integer` and `e=[Char]`)

   ```
   data ExceptionOfState s e a = State (s -> ((Exception e a), s ))
   ```

   the computation always returns a pair value and new state, and the value in this pair can be either an error or a normal value.

Combining monads the hard way

Notice that for the case `State (s -> ((Exception e a), s ))` there are two further possibilities, according to the state we return when an exception is caught. Each possibility corresponds to a different definition of `trywith`

1. backtrack the modifications made by the computation `m` that raised the exception:

```
  trywith m f = \s -> case m s of
                       (Val x , s’) -> (Val x , s’)
                       (Exn x , s’) -> f x s
```

2. keep the modifications made by the computation `m` that raised the exception:

```
  trywith m f = \s -> case m s of
                       (Val x , s’) -> (Val x , s’)
                       (Exn x , s’) -> f x s’
```

Combining monads the hard way

Notice that for the case `State (s -> ((Exception e a), s ))` there are two further possibilities, according to the state we return when an exception is caught. Each possibility corresponds to a different definition of `trywith`

1. backtrack the modifications made by the computation `m` that raised the exception:

```
trywith m f = \s -> case m s of
                      (Val x , s') -> (Val x , s')
                      (Exn x , s') -> f x s
```

2. keep the modifications made by the computation `m` that raised the exception:

```
trywith m f = \s -> case m s of
                      (Val x , s') -> (Val x , s')
                      (Exn x , s') -> f x s'
```

Avoid the boilerplate

Each of the standard monads is specialised to do exactly one thing. In real code, we often need several effects at once. Composing monads by hand or rewriting them from scratch soon reaches its limits

Combining monads by **compositionality**

By applying the monadic transformation to `eval` we passed from a function of type

```
                        Env -> Exp -> Value,
```

to a function of type

```
            Monad m => Env -> Exp -> m Value,
```

In this way we made the code for `eval` parametric in the monad `m`.

Later we chose to instantiate `m` to some particular monad in order to use the specific characteristicts

IDEA: transform the code of an `instance` definition of the monad class so that this definition becomes parametric in some other monad `m`.

Combining monads by **compositionality**

By applying the monadic transformation to `eval` we passed from a function of type

```
Env -> Exp -> Value,
```

to a function of type

```
Monad m => Env -> Exp -> m Value,
```

In this way we made the code for `eval` parametric in the monad `m`.

Later we chose to instantiate `m` to some particular monad in order to use the specific characteristicts

IDEA: transform the code of an `instance` definition of the monad class so that this definition becomes parametric in some other monad `m`.

Monad transformer

A monad instance that is parametric in another monad is a *monad transformer*.

To work on the monad parameter, apply the monadic transformation to the definitions of instances

Monad Transformers

Monad Transformers can help:

- A monad transformer transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, . . .), allowing the programmer to mix and match.
- A form of *aspect-oriented programming*.

Monad Transformers

Monad Transformation in Haskell

- A *monad transformer* maps monads to monads. Represented by a type constructor `T` of the following kind:

```
T ::  (* -> *) -> (* -> *)
```

- Additionally, a monad transformer adds computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:

```
lift ::  M a ->  T M  a
```

Monad Transformers

Monad Transformation in Haskell

- A *monad transformer* maps monads to monads. Represented by a type constructor `T` of the following kind:

```
T ::  (* -> *) -> (* -> *)
```

- Additionally, a monad transformer adds computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:

```
lift ::  M a -> (T M) a
```

Little reminder

Are you lost? ... Let us recap

Goal: write the following code where all the **plumbing** to handle effects is hidden in the definition of `m`

```
eval :: (Monad m) => Env -> Exp -> m Value

eval env (Const i )    = do tick
                            return (Vint i)
eval env (Var n)       = do tick
                            return (env !! n)
eval env (Plus e1 e2) = do tick
                            x1  <- eval env e1
                            x2  <- eval env e2
                            case (x1 , x2) of
                                 (Vint i1, Vint i2)
                                   -> return (Vint (i1 + i2 ))
                                 _ -> raise "type error in addition"
eval env (Abs e)       = do tick
                            return (Vclos env e)
eval env (App e1 e2)  = do tick
                            fun <- eval env e1
                            val <- eval env e2
                            case fun of
                                 Vclos env0 body
                                   -> eval (val : env0 ) body
                                 _ -> raise "type error in application"
```

Are you lost? ... Let us recap

The *dirty work* is in the definition of the monad `m` that will be used. Two ways are possible:

1. **Define `m` from scratch:** Define a new monad `m` so as it combines the effects of the Exception and of the State monads for which `raise` and `tick` are defined.
   Advantages: a fine control on the definition
   Drawbacks: no code reuse, hard to mantain and modify
2. **Define `m` by composition:** Define `m` by composing more elementary blocks that provide functionalities of *states* and *exceptions* respectively.
   Advantages: modular development; in many case it is possible to reuse components from the shelves.
   Drawbacks: Some trade-off since the building blocks may not provide exactly the sought combination of functionalities.

Are you lost? ... Let us recap

The *dirty work* is in the definition of the monad `m` that will be used. Two ways are possible:

1. **Define `m` from scratch:** Define a new monad `m` so as it combines the effects of the Exception and of the State monads for which `raise` and `tick` are defined.
   Advantages: a fine control on the definition
   Drawbacks: no code reuse, hard to mantain and modify
2. **Define `m` by composition:** Define `m` by composing more elementary blocks that provide functionalities of *states* and *exceptions* respectively.
   Advantages: modular development; in many case it is possible to reuse components from the shelves.
   Drawbacks: Some trade-off since the building blocks may not provide exactly the sought combination of functionalities.

Monad transformers

We show the second technique by building the sought `m` from two *monad transformers* for exceptions and states respectively.

Step 1: defining the functionalities

We define two *subclasses* of the `Monad` class

EXCEPTION MONAD

An Exception Monad is a monad with an operation `raise` that takes a string and yields a monadic computation

```
class Monad m => ExMonad m where
  raise :: String -> m a
```

STATE MONAD

A State Monad is a monad with an operation `tick` that yields a computation on values of the unit type.

```
class Monad m => StMonad m where
  tick :: m ()
```

Step 1: defining the functionalities

We define two *subclasses* of the `Monad` class

EXCEPTION MONAD

An Exception Monad is a monad with an operation `raise` that takes a string and yields a monadic computation

```
class Monad m => ExMonad m where
  raise :: String -> m a
```

STATE MONAD

A State Monad is a monad with an operation `tick` that yields a computation on values of the unit type.

```
class Monad m => StMonad m where
  tick :: m ()
```

It is now possible to specify a type for `eval` so that its definition type-checks

```
eval :: (ExMonad m, StMonad m) => Env -> Exp -> m Value

eval env (Const i) = do tick
                           :
                         _ -> raise "type error in addition"
                           :
```

Step 2: defining the building blocks

We now need to define a monad `m` that is an instance of both `StMonad` and `ExMonad`.

We do it by composing two *monad transformers*

Definition (Monad transformer)

A *monad transformer* is a higher-order operator `t` that maps each monad `m` to a monad `(t m)`, equipped with an operation `lift` that promotes a computation `x :: m a` from the original monad `m` that is fed to `t`, to a computation

`(lift x) :: (t m) a`

on the monad `(t m)`.

Definition of the class of monad transformers

```
class MonadTrans t where
  lift :: Monad m => m a -> (t m) a
```

Example

If we want to apply to the monad `Exception String` a transformer `T` that provides some operation `xyz`, then we need to lift `raise` from `Exception String` to `T(Exception String)`.

Without the lifting the only operation defined for `T(Exception String)` would be `xyz`. With `lift` since

```
raise :: String -> Exception String,
```

then:

```
lift.raise :: String -> T(Exception String)
```

Example

If we want to apply to the monad `Exception String` a transformer `T` that provides some operation `xyz`, then we need to lift `raise` from `Exception String` to `T(Exception String)`.

Without the lifting the only operation defined for `T(Exception String)` would be `xyz`. With `lift` since

```
raise :: String -> Exception String,
```

then:

```
lift.raise :: String -> T(Exception String)
```

Nota bene

There is no magic formula to produce the transformer versions of a given monad

Step 2a: A monad transformer for exceptions

Consider again our first monad `Exception e`:

```
data Exception e a = Val a | Exn e

instance Monad (Exception e) where
    return x    = Val x
    m >>= f     = case m of Exn x -> Exn x ; Val x -> f x

raise :: e -> Exception e a
raise x = Exn x
```

Step 2a: A monad transformer for exceptions

Consider again our first monad `Exception e`:

```
data Exception e a = Val a | Exn e

instance Monad (Exception e) where
    return x    = Val x
    m >>= f     = case m of Exn x -> Exn x ; Val x -> f x

raise :: e -> Exception e a
raise x = Exn x
```

We now want to modify the code above in order to obtain a transformer `ExceptionT` in which the computations are themselves on monads, that is:

```
data ExceptionT m a = MkExc (m (Exception String a))
```

The (binary) type constructor `ExceptionT` “puts exceptions inside” another monad `m` (convention: a monad transformers is usually named as the corresponding monad with a ’T’ at the end.)

For the sake of simplicity we consider that exceptions are of type String and not the more general transformer (`ExceptionT e`):

```
data ExceptionT e m a = MkExc (m (Exception e a))
```

Step 2a: A monad transformer for exceptions

Consider again our first monad `Exception e`:

```
data Exception e a = Val a | Exn e

instance Monad (Exception e) where
    return x    = Val x
    m >>= f     = case m of Exn x -> Exn x ; Val x -> f x

raise :: e -> Exception e a
raise x = Exn x
```

We now want to modify the code above in order to obtain a transformer `ExceptionT` in which the computations are themselves on monads, that is:

```
data ExceptionT m a = MkExc (m (Exception String a))
```

The (binary) type constructor `ExceptionT` “puts exceptions inside” another monad `m` (convention: a monad transformers is usually named as the corresponding monad with a ’T’ at the end.)

We want `ExceptionT` to be a *monad transformer*, ie. `(ExceptionT m)` to be a monad: *we must define* `bind` *and* `return` *for the monad* `(ExceptionT m)`:

```
data ExceptionT m a = MkExc (m (Exception String a))

-- The 'recover' function just strips off the outer MkExc constructor,
-- for convenience
recover :: ExceptionT m a -> m (Exception String a)
recover (MkExc x) = x


-- return is easy. It just wraps the value first in the monad m
-- by return (of the underlying monad) and then in MkExc
returnET :: (Monad m) => a -> ExceptionT m a
returnET x = MkExc (return (Val x))


-- A first version for bind uses do and return to work on the
-- underlying monad m ... whatever it is.
bindET :: (Monad m) => (ExceptionT m a) -> ( a -> ExceptionT m b)
                                        ->  ExceptionT m b
bindET (MkExc x) f =                -- x of type m (Exception String a)
              MkExc (               -- we wrap the result in MkExc
                      do y <- x     -- y is of type Exception String a
                         case y of
                           Val z -> recover (f z)
                           Exn z -> return (Exn z) )
```

Notice the use of the monadic syntax (`do`, `return`,. . .) to work on the monad parameter `m`.

Step 2a: A monad transformer for exceptions

More compactly:

```
instance Monad m => Monad (ExceptionT m) where
    return x = MkExc (return (Val x))
    x >>= f  = MkExc (recover x >>= r)
                where r (Exn y) = return (Exn y)
                      r (Val y) = recover (f y)
```

Step 2a: A monad transformer for exceptions

More compactly:

```
instance Monad m => Monad (ExceptionT m) where
    return x = MkExc (return (Val x))
    x >>= f  = MkExc (recover x >>= r)
                where r (Exn y) = return (Exn y)
                      r (Val y) = recover (f y)
```

Moreover, `(ExceptionT m)` is an exception monad, not just a plain one...

```
instance Monad m => ExMonad (ExceptionT m) where
  raise e = MkExc (return (Exn e))
```

Step 2a: A monad transformer for exceptions

More compactly:

```
instance Monad m => Monad (ExceptionT m) where
    return x = MkExc (return (Val x))
    x >>= f  = MkExc (recover x >>= r)
                where r (Exn y) = return (Exn y)
                      r (Val y) = recover (f y)
```

Moreover, `(ExceptionT m)` is an exception monad, not just a plain one...

```
instance Monad m => ExMonad (ExceptionT m) where
  raise e = MkExc (return (Exn e))
```

`ExceptionT` is a monad tranformer because we can lift any action in `m` to an action in `(ExceptionT m)` by wrapping its result in a ’`Val`’ constructor...

```
instance MonadTrans ExceptionT where
    lift g = MkExc $ do { x <- g; return (Val x) }
```

Step 2a: A monad transformer for exceptions

More compactly:

```
instance Monad m => Monad (ExceptionT m) where
    return x = MkExc (return (Val x))
    x >>= f  = MkExc (recover x >>= r)
                where r (Exn y) = return (Exn y)
                      r (Val y) = recover (f y)
```

Moreover, `(ExceptionT m)` is an exception monad, not just a plain one...

```
instance Monad m => ExMonad (ExceptionT m) where
  raise e = MkExc (return (Exn e))
```

`ExceptionT` is a monad tranformer because we can lift any action in `m` to an action in `(ExceptionT m)` by wrapping its result in a ’`Val`’ constructor...

```
instance MonadTrans ExceptionT where
    lift g = MkExc $ do { x <- g; return (Val x) }
```

We can now use the `lift` operation to make `(ExceptionT m)` into a state monad whenever `m` is one, by lifting `m`’s `tick` operation to `(ExceptionT m)`.

```
instance StMonad m => StMonad (ExceptionT m) where
  tick = lift tick
```

Step 2b: A monad transformer for states

```haskell
newtype StateT m a = MkStt ( Int -> m (a,Int))

-- strip off the MkStt constructor
apply :: StateT m a -> Int -> m (a, Int)
apply (MkStt f) = f

-- if m is a monad, then StateT m is a monad
instance Monad m => Monad (StateT m) where
  return x = MkStt $ \s -> return (x,s)
  p >>= q  = MkStt $ \s -> do (x,s’) <- apply p s
                              apply (q x) s’

-- StateT is a monad transformer
instance MonadTrans StateT where
  lift g = MkStt $ \s -> do  x <- g; return (x,s)

-- if m is a monad, then StateT m is not only a monad
-- but also a STATE MONAD
instance (Monad m) => StMonad (StateT m) where
  tick = MkStt $ \s ->  return ((), s+1)

-- use lift to promote StateT m to an exception monad
instance ExMonad m => ExMonad (StateT m) where
  raise e = lift (raise e)
```

Lost again? Let us recap this Step 2

In Step 2 we defined some monad trasformers of the form `XyzT`.

1. To be a "transformer" `XyzT` must map monads into monads. So if `m` is a monad (ie., it provides `bind` and `return`), then so must `(XyzT m)` be. So we define `bind` and `return` for `(XyzT m)` and use monadic notation to work on the generic `m`.
2. But `(XyzT m)` must not only provide `bind` and `return`, but also some operations typical of some class `Xyz`, subclass of the `Monad` class. So we define also these operations by declaring that `(XyzT m)` is an instance of `Xyz`.
3. This is not enough for `XyzT` to be a transformer. It must also provide a `lift` operation. By defining it we declare that `XyzT` is an instance of the class `MonadTrans`
4. Finally we can use the `lift` function to make `(XyzT m)` "inherit" the characteristics of `m`: so if `m` is an instance of some monadic subclass `Abc`, then we can make also `(XyzT m)` be a `Abc` monad simply by lifting (by composition with `lift`) all the operations specific of `Abc`.

Step 3: Putting it all together...

Just a matter of assembling the pieces.

Interestingly, though, there are TWO ways to combine our transformers to build a monad with exceptions and state:

1.
```
evalStEx :: Env -> Exp -> StateT (ExceptionT Identity) Value
evalStEx = eval
```

2.
```
evalExSt ::  Env -> Exp -> ExceptionT (StateT Identity) Value
evalExSt = eval
```

Note that `ExceptionT Identity` and `StateT Identity` are respectively the `Exception` and `State` monads defined before, modulo two modifications:

1. Values are further wrapped in an inner `MkId` constructor
2. To enhance readibility I used distinct names for the types and their constructors, for instance:

   ```
   newtype StateT m a = MkStt (Int -> m (a,Int))
   ```

   rather then

   ```
   newtype StateT m a = StateT (Int -> m (a,Int))
   ```

   as it is customary in the Haskell library

Order matters

At first glance, it appears that `evalExSt` and `evalStEx` do the same thing...

```
five  = (App(Abs(Plus(Var 0)(Const 1)))(Const 4))    --(λx.(x+1))4
wrong = (App(Abs(Plus(Var 0)(Const 1)))(Abs(Var 0))) --(λx.(x+1))(λy.y)

*Main> evalStEx [] five
Vint 5, count: 6

*Main> evalExSt [] five
Vint 5, count: 6
```

Order matters

At first glance, it appears that `evalExSt` and `evalStEx` do the same thing...

```
five  = (App(Abs(Plus(Var 0)(Const 1)))(Const 4))    --(λx.(x+1))4
wrong = (App(Abs(Plus(Var 0)(Const 1)))(Abs(Var 0))) --(λx.(x+1))(λy.y)

*Main> evalStEx [] five
Vint 5, count: 6

*Main> evalExSt [] five
Vint 5, count: 6
```

BUT ...

```
*Main> evalStEx [] wrong
exception: type error in addition

*Main> evalExSt [] wrong
exception: type error in addition, count: 6
```

- `StateT (ExceptionT Identity)` either returns a state or an exception
- `ExceptionT (StateT Identity)` always returns a state

I omitted the code to print the results of monadic computations. It can be found in the accompagnying code: `http://www.irif.fr/~gc/slides/evaluator.hs`

The `Continuation` monad

Computation type: Computations which can be interrupted and resumed.

Binding strategy: Binding a function to a monadic value creates a new continuation which uses the function as the continuation of the monadic computation.

Useful for: Complex control structures, error handling and creating co-routines.

The `Continuation` monad

Computation type: Computations which can be interrupted and resumed.

Binding strategy: Binding a function to a monadic value creates a new continuation which uses the function as the continuation of the monadic computation.

Useful for: Complex control structures, error handling and creating co-routines.

From `haskell.org`:

Abuse of the Continuation monad can produce code that is impossible to understand and maintain.

Many algorithms which require continuations in other languages do not require them in Haskell, due to Haskell's lazy semantics.

The `Continuation` monad

```
newtype Cont r a = Cont ((a -> r) -> r)

app :: Cont r a -> ((a -> r) -> r)          -- remove the wrapping Cont
app (Cont f) = f

instance Monad (Cont r) where
  return a = Cont $ \k -> k a                         -- = λk.k a
  (Cont c) >>= f = Cont $ \k -> c (\a -> app (f a) k) -- = λk.c(λa.f a k)
```

`Cont r a` is a CPS computation that produces an intermediate result of type `a` within a CPS computation whose final result type is `r`.

The return function simply creates a continuation which passes the value on.

The `>>=` operator adds the bound function into the continuation chain.

```
class (Monad m) => MonadCont m where
    callCC :: ((a -> m b) -> m a) -> m a

instance MonadCont (Cont r) where
    callCC f = Cont (\k -> app (f (\a -> Cont (\_ -> k a))) k)
```

Essentially (i.e., without constructors) the definition above states:

`callCC f = λk.fkk`

ie., `f` is like a value but with an extra parameter *k* bound to its current continuatio

No need to define `throw` since we can directly use the continuation by applying it to a value, as shown in the next example

```
bar :: Char -> String -> Cont r String
bar c s = do
  msg <- callCC $ \k -> do
    let s' = c : s
    if (s' == "hello") then k "They say hello." else return ()
    let s'' = show s'
    return ("They appear to be saying " ++ s'')
  return msg
```

When you call `k` with a value, the entire `callCC` call returns that value. In other words, `k` is a 'goto' statement: `k` in our example pops the execution out to where you first called `callCC`, the `msg <- callCC $ ...` line: no more of the argument to `callCC` (the inner do-block) is executed.

No need to define `throw` since we can directly use the continuation by applying it to a value, as shown in the next example

```
bar :: Char -> String -> Cont r String
bar c s = do
  msg <- callCC $ \k -> do
    let s’ = c : s
    if (s’ == "hello") then k "They say hello." else return ()
    let s’’ = show s’
    return ("They appear to be saying " ++ s’’)
  return msg
```

When you call `k` with a value, the entire `callCC` call returns that value. In other words, `k` is a ’goto’ statement: `k` in our example pops the execution out to where you first called `callCC`, the `msg <- callCC $ ...` line: no more of the argument to `callCC` (the inner do-block) is executed. This is shown by two different executions, to which we pass the function `print` as continuation:

```
main = do
     app (bar ’h’ "ello") print
     app (bar ’h’ "llo.") print
```

Which once compiled and executed produces the following output

```
"They say hello."
"They appear to be saying \"hllo.\""
```

A simpler example is the following one which contains a useless line:

```
bar :: Cont r Int
bar = callCC $ \k -> do
  let n = 5
  k n
  return 25
```

`bar` will always return `5`, and never `25`, because we pop out of `bar` before getting to the `return 25` line.

Summary

Purity has advantages but effects are unavoidable.

- To have them both, effects must be explicitly programmed.
- In order to separate the definition of the algorithm from the definition of the plumbing that manages the effects it is possible to use a monad. The monad centralizes all the programming that concerns effects.
- Several effects may be necessary in the same program. One can define the corresponding monad by composing monad transformers. These are functions from monads to monads, each handling a specific effect.

Summary

Purity has advantages but effects are unavoidable.

- To have them both, effects must be explicitly programmed.
- In order to separate the definition of the algorithm from the definition of the plumbing that manages the effects it is possible to use a monad. The monad centralizes all the programming that concerns effects.
- Several effects may be necessary in the same program. One can define the corresponding monad by composing monad transformers. These are functions from monads to monads, each handling a specific effect.

However

- Putting code in monadic form is easy and can be done automatically, but there is no magic formula to define monads or even derive from given monads the corresponding trasformers
- Understanding monadic code is relatively straightforward but writing and debugging monads or monads transformers from scracth may be dreadful.

Summary

Purity has advantages but effects are unavoidable.

- To have them both, effects must be explicitly programmed.
- In order to separate the definition of the algorithm from the definition of the plumbing that manages the effects it is possible to use a monad. The monad centralizes all the programming that concerns effects.
- Several effects may be necessary in the same program. One can define the corresponding monad by composing monad transformers. These are functions from monads to monads, each handling a specific effect.

However

- Putting code in monadic form is easy and can be done automatically, but there is no magic formula to define monads or even derive from given monads the corresponding trasformers
- Understanding monadic code is relatively straightforward but writing and debugging monads or monads transformers from scracth may be dreadful.

Suggestion

Use **existing** monads and monads trasformers as much as possible.

Outline

Monads and ML Functors

- Monads define the bind and return functions that are the core of the plumbing of effects
- Specific operations for effects such as `raise` and `tick` are provided by subclasses of Monads (eg, `StMonad`, `ExMonad`).
- Modular development is obtained by *monad transformers* which are functions from monads to (subclasses of) monads.

We can reproduce monads by modules and transformers by functors.

Signature for monads

The Caml module signature for a monad is:

```
module type MONAD = sig
    type α mon
    val return: α -> α mon
    val bind: α mon -> (α -> β mon) -> β mon
end
```

The Identity monad

The Identity monad is a trivial instance of this signature:

```
module Identity = struct
    type α mon = α
    let return x = x
    let bind m f = f m
end
```

Monad Transformers

Monad transformer for exceptions

```
module ExceptionT(M: MONAD) = struct
  type α outcome = Val of α | Exn of exn
  type α mon = (α outcome) M.mon
  let return x = M.return (Val x)
  let bind m f =
    M.bind m (function Exn e -> M.return (Exn e) | Val v -> f v)
  let lift x = M.bind x (fun v -> M.return (Val v))
  let raise e = M.return (Exn e)
  let trywith m f =
    M.bind m (function Exn e -> f e | Val v -> M.return (Val v))
end
```

Monad Transformers

Monad transformer for exceptions

```
module ExceptionT(M: MONAD) = struct
  type α outcome = Val of α | Exn of exn
  type α mon = (α outcome) M.mon
  let return x = M.return (Val x)
  let bind m f =
    M.bind m (function Exn e -> M.return (Exn e) | Val v -> f v)
  let lift x = M.bind x (fun v -> M.return (Val v))
  let raise e = M.return (Exn e)
  let trywith m f =
    M.bind m (function Exn e -> f e | Val v -> M.return (Val v))
end
```

Notice the lesser flexibility due to the lack of the (static) overloading (provided by Haskell's type-classes) which obliges us to specify whose `bind` and `return` we use.

Monad Transformers

Monad transformer for exceptions

```
module ExceptionT(M: MONAD) = struct
  type α outcome = Val of α | Exn of exn
  type α mon = (α outcome) M.mon
  let return x = M.return (Val x)
  let bind m f =
    M.bind m (function Exn e -> M.return (Exn e) | Val v -> f v)
  let lift x = M.bind x (fun v -> M.return (Val v))
  let raise e = M.return (Exn e)
  let trywith m f =
    M.bind m (function Exn e -> f e | Val v -> M.return (Val v))
end
```

Notice the lesser flexibility due to the lack of the (static) overloading (provided by Haskell's type-classes) which obliges us to specify whose `bind` and `return` we use.

Also the fact that the `ExceptionT` functor returns a module that is (1) a *monad* (2) an instance of the *exception monad*, and (3) a *transformer*, is lost in the definition of the functions exported by the module [(1) holds because of `bind` and `return`, (2) because of `raise` and `trywith`, and (3) because of `lift`]

Monad Transformers

Monad transformer for state

```
module StateT(M: MONAD) = struct
  type α mon = state -> (α * state) M.mon
  let return x = fun s -> M.return (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.return (x, s))
  let ref x = fun s -> M.return (store_alloc x s)
  let deref r = fun s -> M.return (store_read r s, s)
  let assign r x = fun s -> M.return (store_write r x s)
end
```

Using monad transformers

```
module State = StateT(Identity)

module StateAndException = struct
      include ExceptionT(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 $\rightarrow$ α outcome $\times$ state, i.e. state is preserved when raising exceptions. The other combination, StateT(ExceptionT(Identity)) gives α mon = state $\rightarrow$ (α $\times$ state) outcome, i.e. state is discarded when an exception is raised.

Exercise

Define the functor for continuation monad transformer.

```
module ContTransf(M: MONAD) = struct
  type α mon = (α -> answer M.mon) -> answer M.mon
  let return x =
  let bind m f =
  let lift m =

  let callcc f =
  let throw c x =
end
```

Exercise

Define the functor for continuation monad transformer.

```
module ContTransf(M: MONAD) = struct
  type α mon = (α -> answer M.mon) -> answer M.mon
  let return 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 callcc f =  fun k -> f k k
  let throw c x =  fun k -> c x
end
```

References

- Philip Wadler. Monads for functional Programming. In Advanced Functional Programming, Proceedings of the Baastad Spring School, Lecture Notes in Computer Science n. 925, Springer, 1995.
- Martin Grabmüller. Monad Transformers Step by Step, Unpublished draft. 2006 `http://www.grabmueller.de/martin/www/pub/`