Monads

Fast Track to Haskell

Andres Löh, Edsko de Vries

8–9 April 2013 —

Well-Typed
The Haskell Consultants

The plan

Let us look at a number of datatypes and typical programming problems involving these types . . .

The Maybe type

```
data Maybe a = Nothing
             | Just a
```

The Maybe datatype is often used to encode failure or an exceptional value:

```
lookup :: (Eq a) ⇒ a → [(a, b)] → Maybe b
find   :: (a → Bool) → [a] → Maybe a
```


Encoding exceptions using Maybe

Assume that we have a data structure with the following operations:

```
up, down, right :: Loc → Maybe Loc
update          :: (Int → Int) → Loc → Loc
```

Given a location l_1, we want to move up, right, down, and update the resulting position with using update (+ 1) ...

Each of the steps can fail.

Encoding exceptions using Maybe (contd.)

```
case up l1 of
  Nothing → Nothing
  Just l2  → case right l2 of
               Nothing → Nothing
               Just l3  → case down l3 of
                            Nothing → Nothing
                            Just l4  → Just (update (+ 1) l4)
```

In essence, we need

- a way to sequence function calls and use their results if successful
- a way to modify or produce successful results.

Encoding exceptions using Maybe (contd.)

```
case up l1 of
  Nothing → Nothing
  Just l2  → case right l2 of
               Nothing → Nothing
               Just l3  → case down l3 of
                            Nothing → Nothing
                            Just l4  → Just (update (+ 1) l4)
```

Sequencing:

```
(>>=) :: Maybe a → (a → Maybe b) → Maybe b
f >>= g = case f of
            Nothing → Nothing
            Just x  → g x
```

Encoding exceptions using Maybe (contd.)

$$
\begin{array}{l}
\mathsf{up}\ l_1 \ggg= \\
\quad \lambda\, l_2 \quad \to \mathsf{right}\ l_2 \ggg= \\
\qquad\qquad \lambda\, l_3 \quad \to \mathsf{down}\ l_3 \ggg= \\
\qquad\qquad\qquad\qquad \lambda\, l_4 \quad \to \mathsf{Just}\ (\mathsf{update}\ (+\ 1)\ l_4)
\end{array}
$$

Sequencing:

$$
\begin{array}{l}
(\ggg=) :: \mathsf{Maybe}\ a \to (a \to \mathsf{Maybe}\ b) \to \mathsf{Maybe}\ b \\
f \ggg= g = \mathbf{case}\ f\ \mathbf{of} \\
\qquad\qquad \mathsf{Nothing} \to \mathsf{Nothing} \\
\qquad\qquad \mathsf{Just}\ x \quad \to g\ x
\end{array}
$$

Sequencing and embedding

$$
\begin{array}{l}
\mathsf{up}\ l_1 \ggg= \\
\quad \lambda l_2 \to \mathsf{right}\ l_2 \ggg= \\
\qquad\quad \lambda l_3 \to \mathsf{down}\ l_3 \ggg= \\
\qquad\qquad\quad \lambda l_4 \to \boxed{\mathsf{Just}\ (\mathsf{update}\ (+\ 1)\ l_4)}
\end{array}
$$

$$
\begin{array}{l}
(\ggg=) :: \mathsf{Maybe}\ a \to (a \to \mathsf{Maybe}\ b) \to \mathsf{Maybe}\ b \\
f \ggg= g \quad = \mathbf{case}\ f\ \mathbf{of} \\
\qquad\qquad \mathsf{Nothing} \to \mathsf{Nothing} \\
\qquad\qquad \mathsf{Just}\ x \quad \to g\ x \\
\mathsf{return} :: a \to \mathsf{Maybe}\ a \\
\mathsf{return}\ x = \mathsf{Just}\ x
\end{array}
$$

$$
(\mathsf{up}\ l_1) \ggg= \mathsf{right} \ggg= \mathsf{down} \ggg= \mathsf{return} \circ \mathsf{update}\ (+\ 1)
$$

Observation

Code looks a bit like imperative code. Compare:

$$
\begin{array}{ll}
\mathsf{up}\ \mathsf{l}_1 & \ggg\!= \lambda \mathsf{l}_2 \rightarrow \\
\mathsf{right}\ \mathsf{l}_2 & \ggg\!= \lambda \mathsf{l}_3 \rightarrow \\
\mathsf{down}\ \mathsf{l}_3 & \ggg\!= \lambda \mathsf{l}_4 \rightarrow \\
\multicolumn{2}{l}{\mathsf{return}\ (\mathsf{update}\ (+\ 1)\ \mathsf{l}_4)}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{l}_2 := \mathsf{up}\ \mathsf{l}_1; \\
\mathsf{l}_3 := \mathsf{right}\ \mathsf{l}_2; \\
\mathsf{l}_4 := \mathsf{down}\ \mathsf{l}_3; \\
\textbf{return}\ \mathsf{update}\ (+\ 1)\ \mathsf{l}_4
\end{array}
$$

- In the imperative language, the occurrence of possible exceptions is a side effect.
- Haskell is more explicit because we use the Maybe type and the appropriate sequencing operation.

A variation: Either

Compare the datatypes

$$
\begin{array}{ll}
\textbf{data}\ \mathsf{Either}\ \mathsf{a}\ \mathsf{b} & = \mathsf{Left}\ \mathsf{a} \mid \mathsf{Right}\ \mathsf{b} \\
\textbf{data}\ \mathsf{Maybe}\ \mathsf{a} & = \mathsf{Nothing} \mid \mathsf{Just}\ \mathsf{a}
\end{array}
$$

The datatype Maybe can encode exceptional function results (i.e., failure), but no information can be associated with Nothing. We cannot dinstinguish different kinds of errors.

Using Either, we can use Left to encode errors, and Right to encode successful results.

Example

```
type Error = String

fac :: Int → Either Error Int
fac 0              = Right 1
fac n | n > 0      = case fac (n − 1) of
                       Left  e → Left e
                       Right r → Right (n ∗ r)
      | otherwise  = Left "fac: negative argument"
```

Structure of sequencing looks similar to the sequencing for Maybe .

Sequencing and returning for Either

We can define variations of the operatons for Maybe :

```
(≫=) :: Either Error a → (a → Either Error b) → Either Error b
f ≫= g = case f of
           Left  e → Left e
           Right x → g x

return :: a → Either Error a
return x = Right x
```

The function can now be written as:

```
fac :: Int → Either Error Int
fac 0              = return 1
fac n | n > 0      = fac (n − 1) ≫= λr → return (n ∗ r)
      | otherwise  = Left "fac: negative argument"
```

Simulating exceptions

We can abstract completely from the definition of the underlying `Either` type if we define functions to throw and catch errors.

```
throwError :: Error → Either Error a
throwError e = Left e

catchError :: Either Error a →          -- computation
   (Error → Either Error a) →           -- handler
   Either Error a
catchError f handler = case f of
                         Left   e → handler e
                         Right x → Right x
```


Maintaining state explicitly

- We pass state to a function as an argument.
- The function modifies the state and produces it as a result.
- If the function does anything except modifying the state, we must return a tuple (or a special-purpose datatype with multiple fields).

This motivates the following type definition:

```
type State s a = s → (a, s)
```

Using state

There are many situations where maintaining state is useful:

- using a random number generator

```haskell
type Random a = State StdGen a
```

- using a counter to generate unique labels

```haskell
type Counter a = State Int a
```

- maintaining the complete current configuration of an application (an interpreter, a game, ...) using a user-defined datatype

```haskell
data ProgramState = ...
type Program a = State ProgramState a
```


Example: labelling the leaves of a tree

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```haskell
labelTree :: Tree a -> State Int (Tree (a, Int))
labelTree (Leaf x)   c  = (Leaf (x, c), c + 1)
labelTree (Node l r) c1 = let (ll, c2) = labelTree l c1
                              (lr, c3) = labelTree r c2
                          in (Node ll lr, c3)
```

Encoding state passing

$$
\begin{aligned}
\lambda s_1 \to \textbf{let}\ & (\boxed{\mathsf{lvl}}\ , s_2) = \boxed{\mathsf{generateLevel}} && s_1 \\
& (\boxed{\mathsf{lvl}'}, s_3) = \boxed{\mathsf{generateStairs}\ \mathsf{lvl}} && s_2 \\
& (\boxed{\mathsf{ms}}, s_4) = \boxed{\mathsf{placeMonsters}\ \mathsf{lvl}'} && s_3 \\
\textbf{in}\ & (\boxed{\mathsf{combine}\ \mathsf{lvl}'\ \mathsf{ms}}, s_4)
\end{aligned}
$$

Encoding state passing

$$
\begin{aligned}
\lambda s_1 \to \textbf{let}\ & (\mathsf{lvl}\ , s_2) = \mathsf{generateLevel} && s_1 \\
& (\mathsf{lvl}', s_3) = \mathsf{generateStairs}\ \mathsf{lvl} && s_2 \\
& (\mathsf{ms}, s_4) = \mathsf{placeMonsters}\ \mathsf{lvl}' && s_3 \\
\textbf{in}\ & (\mathsf{combine}\ \mathsf{lvl}'\ \mathsf{ms}, s_4)
\end{aligned}
$$

Again, we need

- a way to sequence function calls and use their results
- a way to modify or produce successful results.

Encoding state passing

$$
\begin{aligned}
\lambda s_1 \rightarrow \textbf{let}\ & (lvl\ \ , s_2) = generateLevel\ \ \ \ \ \ \ s_1 \\
& (lvl', s_3) = generateStairs\ lvl\ \ s_2 \\
& (ms, s_4) = placeMonsters\ lvl'\ s_3 \\
\textbf{in}\ & (combine\ lvl'\ ms, s_4)
\end{aligned}
$$

$$
\begin{aligned}
&(\gg=) :: State\ s\ a \rightarrow (a \rightarrow State\ s\ b) \rightarrow State\ s\ b \\
&f \gg= g \quad = \lambda s \rightarrow \textbf{let}\ (x, s') = f\ s\ \textbf{in}\ g\ x\ s' \\
&return :: a \rightarrow State\ s\ a \\
&return\ x = \lambda s \rightarrow (x, s)
\end{aligned}
$$

Bind and return for state

$$
\begin{aligned}
& \quad generateLevel \quad\quad\quad \gg= \lambda lvl \rightarrow \\
& \quad generateStairs\ lvl \quad \gg= \lambda lvl' \rightarrow \\
& \quad placeMonsters\ lvl' \quad \gg= \lambda ms \rightarrow \\
& return\ (combine\ lvl'\ ms)
\end{aligned}
$$

$$
\begin{aligned}
&(\gg=) :: State\ s\ a \rightarrow (a \rightarrow State\ s\ b) \rightarrow State\ s\ b \\
&f \gg= g \quad = \lambda s \rightarrow \textbf{let}\ (x, s') = f\ s\ \textbf{in}\ g\ x\ s' \\
&return :: a \rightarrow State\ s\ a \\
&return\ x = \lambda s \rightarrow (x, s)
\end{aligned}
$$

Observation

Again, the code looks a bit like imperative code. Compare:

$$\begin{array}{ll}\mathsf{generateLevel} & \gg\!= \lambda \mathsf{lvl} \rightarrow \\ \mathsf{generateStairs\ lvl} & \gg\!= \lambda \mathsf{lvl}' \rightarrow \\ \mathsf{placeMonsters\ lvl}' & \gg\!= \lambda \mathsf{ms} \rightarrow \\ \multicolumn{2}{l}{\mathsf{return\ (combine\ lvl'\ ms)}}\end{array}$$

$$\begin{array}{ll}\mathsf{lvl} & := \mathsf{generateLevel}; \\ \mathsf{lvl}' & := \mathsf{generateStairs\ lvl}; \\ \mathsf{ms} & := \mathsf{placeMonsters\ lvl}'; \\ \multicolumn{2}{l}{\mathbf{return}\ \mathsf{combine\ lvl'\ ms}}\end{array}$$

- In the imperative language, the occurrence of memory updates (random numbers) is a side effect.
- Haskell is more explicit because we use the State type and the appropriate sequencing operation.

"Primitive" operations for state handling

We can completely hide the implementation of State if we provide the following two operations as an interface:

$$\begin{array}{l}\mathsf{get} :: \mathsf{State\ s\ s} \\ \mathsf{get} = \lambda \mathsf{s} \rightarrow (\mathsf{s}, \mathsf{s}) \\ \mathsf{put} :: \mathsf{s} \rightarrow \mathsf{State\ s\ ()} \\ \mathsf{put\ s} = \lambda\_ \rightarrow ((), \mathsf{s})\end{array}$$

$$\begin{array}{l}\mathsf{inc} :: \mathsf{State\ Int\ ()} \\ \mathsf{inc} = \mathsf{get} \gg\!= \lambda \mathsf{s} \rightarrow \mathsf{put\ (s + 1)}\end{array}$$

Labelling a tree, revisited

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
labelTree :: Tree a → State Int (Tree (a, Int))
labelTree (Leaf x)   = get >>= λc → inc >> return (Leaf (x, c))
labelTree (Node l r) = labelTree l >>= λll →
                       labelTree r >>= λlr →
                       return (Node ll lr)
```

New version, with implicit state passing, yet explicit sequencing.

```
(>>) :: State s a → State s b → State s b
x >> y = x >>= λ_ → y
```

(The same definition as for IO . . .)

Encoding multiple results and nondeterminism

Get the length of all words in a list of multi-line texts:

```
map length (concat (map words (concat (map lines txts))))
```

What is a notion of embedding and sequencing for computations with many results (nondeterministic computations)?

- Embedding a normal computation into a nondeterminstic one can work by saying the computation has exactly one result.
- Sequencing operations can work by performing the second computation on all possible results of the first one.

Defining bind and return for lists

$(\gg=) :: [a] \to (a \to [b]) \to [b]$
$xs \gg= f = concat\ (map\ f\ xs)$

$return :: a \to [a]$
$return\ x = [x]$

Note that we have to use concat in $(\gg=)$ to flatten the list of lists.

Using bind and return for lists

$map\ length\ (concat\ (map\ words\ (concat\ (map\ lines\ txts))))$

$$\begin{array}{l} txts \quad \gg= \lambda t \to \\ lines\ t \gg= \lambda l \to \\ words\ l \gg= \lambda w \to \\ return\ (length\ w) \end{array} \qquad \begin{array}{l} t := txts \\ l := lines\ t \\ w := words\ w \\ \textbf{return}\ length\ w \end{array}$$

- Again, we have a similarity to imperative code.
- In the imperative language, we have implicit nondeterminism (one or all of the options are chosen).
- In Haskell, we are explicit by using the list datatype and explicit sequencing using $(\gg=)$.

Intermediate Summary

At least four types with $(\gg=)$ and `return`:

- for `Maybe`, $(\gg=)$ sequences operations that may fail and shortcuts evaluation once failure occurs; `return` embeds a function that never fails;
- for `State`, $(\gg=)$ sequences operations that may modify some state and threads the state through the operations; `return` embeds a function that never modifies the state;
- for `[]`, $(\gg=)$ sequences operations that may have multiple results and executes subsequent operations for each of the previous results; `return` embeds a function that only ever has one result.
- for `IO`, $(\gg=)$ sequences the side effects to the outside world, and `return` embeds a function without any side effects.

There is a common interface here!

`Monad` class

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

- The name “monad” is borrowed from category theory.
- A monad is an algebraic structure similar to a monoid.
- Monads have been popularized in functional programming via the work of Moggi and Wadler.

Instances

```
instance Monad Maybe where
  ...
instance (Error e) ⇒ Monad (Either e) where
  ...
instance Monad [] where
  ...
newtype State s a = State {runState :: s → (a, s)}
instance Monad (State s) where
  ...
```

The **newtype** for State is required because Haskell does not allow us to directly make a type $s \to (a, s)$ an instance of Monad. (Question: why not?)

There are more monads

The types we have seen: Maybe, Either, [], State, IO are among the most frequently used monads – but there are many more you will encounter sooner or later.

In fact, we have already seen one more! Which one?

The generators Gen from QuickCheck form a monad. You can see it as an abstract state monad, allowing access to the state of a random number generator.

Additional monad operations

Class Monad contains two additional methods, but with default methods:

```
class Monad m where
  ...
  (>>) :: m a → m b → m b
  m >> n = m >>= λ_ → n
  fail :: String → m a
  fail s = error s
```

While the presence of (>>) can be justified for efficiency reasons, the presence of fail is often considered to be a design mistake.

do notation

The do notation we have introduced when discussing IO is available for all monads:

```
generateLevel         >>= λlvl  →
generateStairs lvl    >>= λlvl' →
placeMonsters lvl'    >>= λms   →
return (combine lvl' ms)
```

```
do
  lvl  ← generateLevel
  lvl' ← generateStairs lvl
  ms   ← placeMonsters lvl'
  return (combine lvl' ms)
```

do notation – contd.

$$\begin{array}{l}
\text{up } l_1 \gg\!= \lambda l_2 \rightarrow \\
\text{right } l_2 \gg\!= \lambda l_3 \rightarrow \\
\text{down } l_3 \gg\!= \lambda l_4 \rightarrow \\
\text{return (update } (+\ 1)\ l_4)
\end{array}$$

$$\begin{array}{l}
\textbf{do} \\
\quad l_2 \leftarrow \text{up } l_1 \\
\quad l_3 \leftarrow \text{right } l_2 \\
\quad l_4 \leftarrow \text{down } l_3 \\
\quad \text{return (update } (+\ 1)\ l_4)
\end{array}$$

Tree labelling, revisited once more

Using Control.Monad.State and **do** notation:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
labelTree :: Tree a → State Int (Tree (a, Int))
labelTree (Leaf x)   = do
  c ← get
  put (c + 1)   -- or  modify (+ 1)
  return (Leaf (x, c))
labelTree (Node l r) = do
  ll ← labelTree l
  lr ← labelTree r
  return (Node ll lr)
```

How to get at the final tree?

Running a stateful computation

```
evalState :: State s a → s → a
```

```
labelTreeFrom0 :: Tree a → Tree (a, Int)
labelTreeFrom0 t = evalState (labelTree t) 0
```

There's also

```
runState :: State s a → s → (a, s)
```

(which is just unpacking `State`'s **newtype** wrapper).

List comprehensions

For list computations

```
map length (concat (map words (concat (map lines txts))))
```

we can use **do** notation

```
do
  t ← txts
  l ← lines t
  w ← words l
  return (length w)
```

but also list comprehensions:

```
[length w | t ← txts, l ← lines t, w ← words l]
```

More on do notation (and list comprehensions)

- Use it, the special syntax is usually more concise.
- Never forget that it is just syntactic sugar. Use $(\gg\!=)$ and $(\gg)$ directly when it is more convenient.

And some things I've already said about IO:

- Remember that return is just a normal function:
  - Not every do-block ends with a return.
  - return can be used in the middle of a do-block, and it doesn't "jump" anywhere.
- Not every monad computation has to be in a do-block. In particular do e is the same as e.
- On the other hand, you may have to "repeat" the do in some places, for instance in the branches of an if.

The IO monad is special

- IO is a primitive type, and $(\gg\!=)$ and return for IO are primitive functions,
- there is no (politically correct) function runIO :: IO a $\rightarrow$ a, whereas for most other monads there is a corresponding function, or at least some way to get an a out of the monad;
- values of IO a denote side-effecting programs that can be executed by the run-time system.

Effectful programming

- IO being special has little to do with it being a monad;
- you can use IO an functions on IO very much ignoring the presence of the Monad class;
- IO is about allowing real side effects to occur; the other types we have seen are entirely pure as far as Haskell is concerned, even though the capture a form of effects.

IO, internally

If you ask GHCi about IO by saying `:i IO`, you get

```
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
      → (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
    -- Defined in ‘ GHC.Types ’
```

So internally, GHC models IO as a kind of state monad having the “real world” as state!

The advantages of an abstract interface

There are several advantages to identifying the “monad” interface:

- We have to learn fewer names. We can use the same `return` and `(>>=)` (and **do** notation) in many different situations.
- There are all sorts of useful derived functions that only use `return` and `(>>=)`. All these library functions become automatically available for every monad now.
- There are many more monads than the ones we've discusses so far. Monads can be combined to form new monads.
- Application-specific code often uses just the monadic interface plus a few extra functions. As such, it is easy to switch the underlying monad of a large part of a program in order to accommodate a new aspect (error handling, logging, backtracking, ...).

Useful monad operations

```
liftM      :: (a → b) → IO a → IO b
mapM       :: (a → IO b) → [a] → IO [b]
mapM_      :: (a → IO b) → [a] → IO ()
forM       :: [a] → (a → IO b) → IO [b]
forM_      :: [a] → (a → IO b) → IO ()
sequence   :: [IO a] → IO [a]
sequence_  :: [IO a] → IO ()
forever    :: IO a → IO b
filterM    :: (a → IO Bool) → [a] → IO [a]
replicateM :: Int → IO a → IO [a]
replicateM_ :: Int → IO a → IO ()
when       :: Bool → IO () → IO ()
unless     :: Bool → IO () → IO ()
```

We had discussed these functions in the context of `IO`.

Useful monad operations

```
liftM       :: Monad m ⇒ (a → b) → m a → m b
mapM        :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM_       :: Monad m ⇒ (a → m b) → [a] → m ()
forM        :: Monad m ⇒ [a] → (a → m b) → m [b]
forM_       :: Monad m ⇒ [a] → (a → m b) → m ()
sequence    :: Monad m ⇒ [m a] → m [a]
sequence_   :: Monad m ⇒ [m a] → m ()
forever     :: Monad m ⇒ a → m b
filterM     :: Monad m ⇒ (a → m Bool) → [a] → m [a]
replicateM  :: Monad m ⇒ Int → m a → m [a]
replicateM_ :: Monad m ⇒ Int → m a → m ()
when        :: Monad m ⇒ Bool → m () → m ()
unless      :: Monad m ⇒ Bool → m () → m ()
```

They're actually all overloaded! Try to infer what each of these mean for `Maybe`, `State` and `[]`.

Example: labelling a rose tree

```
data Rose a = Fork a [Rose a]
```

Each node has a (possibly empty) list of subtrees.

```
labelRose :: Rose a → State Int (Rose (a, Int))
labelRose (Fork x cs) = do
  c ← get
  put (c + 1)
  lcs ← mapM labelRose cs
  return (Fork (x, c) lcs)
```

Questions

What do you think these will evaluate to:

```
replicateM 2 [1..3]
mapM return [1..3]
sequence [[1,2],[3,4],[5,6]]
mapM (flip lookup [(1,'x'),(2,'y'),(3,'z')]) [1..3]
mapM (flip lookup [(1,'x'),(2,'y'),(3,'z')]) [1,4,3]
evalState (replicateM_ 5 (modify (+2)) >> get) 0
```


About liftM and fmap

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
fmap  :: (Functor f) => (a -> b) -> f a -> f b
```

- Nearly same type as fmap, but a different class constraint.
- For historic reasons, Functor is not a superclass of Monad in Haskell.
- But every monad can be made an instance of Functor, by defining fmap to be liftM.
- In practice, nearly all Haskell monads provide a Functor instance. So you usually have liftM, fmap and (<$>) available, all doing the same.

A common pattern

Let's once again look at tree labelling:

$$
\begin{array}{l}
\mathsf{labelTree} :: \mathsf{Tree}\ \mathsf{a} \rightarrow \mathsf{State}\ \mathsf{Int}\ (\mathsf{Tree}\ (\mathsf{a}, \mathsf{Int})) \\
\mathsf{labelTree}\ (\mathsf{Leaf}\ \mathsf{x}) \quad = \mathbf{do} \\
\quad \mathsf{c} \leftarrow \mathsf{get} \\
\quad \mathsf{put}\ (\mathsf{c} + 1) \quad \text{-- or}\ \ \mathsf{modify}\ (+\ 1) \\
\quad \mathsf{return}\ (\mathsf{Leaf}\ (\mathsf{x}, \mathsf{c})) \\
\mathsf{labelTree}\ (\mathsf{Node}\ \mathsf{l}\ \mathsf{r}) = \mathbf{do} \\
\quad \boxed{\begin{array}{l} \mathsf{ll} \leftarrow \mathsf{labelTree}\ \mathsf{l} \\ \mathsf{lr} \leftarrow \mathsf{labelTree}\ \mathsf{r} \\ \mathsf{return}\ (\mathsf{Node}\ \mathsf{ll}\ \mathsf{lr}) \end{array}}
\end{array}
$$

We are returning an application of (constructor) function Node to the results of monadic computations.

A common pattern (contd.)

$$
\begin{array}{l}
\mathbf{do} \\
\quad \mathsf{r}_1 \leftarrow \mathsf{comp}_1 \\
\quad \mathsf{r}_2 \leftarrow \mathsf{comp}_2 \\
\quad \ldots \\
\quad \mathsf{r}_n \leftarrow \mathsf{comp}_n \\
\quad \mathsf{return}\ (\mathsf{f}\ \mathsf{r}_1\ \mathsf{r}_2 \ldots \mathsf{r}_n)
\end{array}
$$

This isn't type correct:

$$\mathsf{f}\ \mathsf{comp}_1\ \mathsf{comp}_2 \ldots \mathsf{comp}_n$$

But we can get close:

$$\mathsf{f} <\$> \mathsf{comp}_1 <*> \mathsf{comp}_2 \ldots <*> \mathsf{comp}_n$$

Monadic application

We need a function that's like function application, but works on monadic values:

```haskell
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf mx = do
  f <- mf
  x <- mx
  return (f x)
```

Types supporting `return` and `ap` have their own name:

```haskell
class Functor f => Applicative f where
  pure  :: a -> f a                    -- like return
  (<*>) :: f (a -> b) -> f a -> f b    -- like ap
```

Every monad can be made into an applicative functor using the obvious **instance** definition.

Example

```haskell
labelTree :: Tree a -> State Int (Tree (a, Int))
labelTree (Leaf x)   = do
  c <- get
  put (c + 1)    -- or modify (+ 1)
  return (Leaf (x, c))
labelTree (Node l r) = Node <$> labelTree l <*> labelTree r
```

Exercise: Convince yourself that this is type correct.

Lessons

- The abstraction of monads is useful for a multitude of different types.
- Monads can be seen as tagging computations with effects.
- While IO is impure and cannot be defined in Haskell, the other effects we have seen can be modelled in a pure way:
  - exceptions via Maybe or Either ;
  - state via State ;
  - nondeterminism via [] .
- The monad interface offers a large number of useful abstractions that can all be applied to these different scenarios.
- All monads are also applicative functors and in particular functors. The $(<\$>)$ and $(<*>)$ operations are also useful for structuring effectful code in Haskell.