A Blessing and a Curse
A Blessing and a Curse

• The **BIG** advantage of *pure* functional programming is
A Blessing and a Curse

• The **BIG** advantage of *pure* functional programming is
  
  “everything is explicit;”

  i.e., flow of data manifest, no side effects.
A Blessing and a Curse

- The **BIG** advantage of pure functional programming is “everything is explicit;” i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.
A Blessing and a Curse

- The **BIG** advantage of *pure* functional programming is "everything is explicit;" i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.
- The **BIG** problem with *pure* functional programming is
A Blessing and a Curse

- The **BIG** advantage of *pure* functional programming is
  
  "everything is explicit;"

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

- The **BIG** problem with *pure* functional programming is
  
  "everything is explicit."
A Blessing and a Curse

• The **BIG** advantage of **pure** functional programming is
  
  “everything is explicit;”

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

• The **BIG** problem with **pure** functional programming is
  
  “everything is explicit.”

  Can add a lot of clutter, make it hard to maintain code
Conundrum

“Shall I be pure or impure?” (Wadler, 1992)
Conundrum

“Shall I be pure or impure?” (Wadler, 1992)

- Absence of effects
  - facilitates understanding and reasoning
  - makes lazy evaluation viable
  - allows choice of reduction order, e.g. parallel
  - enhances modularity and reuse.
Conundrum

“Shall I be pure or impure?” (Wadler, 1992)

• Absence of effects
  - facilitates understanding and reasoning
  - makes lazy evaluation viable
  - allows choice of reduction order, e.g. parallel
  - enhances modularity and reuse.

• Effects (state, exceptions, . . . ) can
  - help making code concise
  - facilitate maintenance
  - improve the efficiency.
**Example: A Compiler Fragment (1)**

*Identification* is the task of relating each applied identifier occurrence to its declaration or definition:

```java
public class C {
    int x, n;
    void set(int n) { x = n; }
}
```
Identification is the task of relating each applied identifier occurrence to its declaration or definition:

```java
public class C {
    int x, n;
    void set(int n) { x = n; }
}
```

In the body of `set`, the one applied occurrence of
- `x` refers to the **instance variable** `x`
**Identification** is the task of relating each applied identifier occurrence to its declaration or definition:

```java
public class C {
    int x, n;
    void set(int n) { x = n; }
}
```

In the body of `set`, the one applied occurrence of

- `x` refers to the *instance variable* `x`
- `n` refers to the *argument* `n`. 
Consider an AST $\text{Exp}$ for a simple expression language. $\text{Exp}$ is a parameterized type: the type parameter $a$ allows variables to be annotated with an attribute of type $a$.

```
data Exp a = LitInt Int |
  Var Id a |
  UnOpApp UnOp (Exp a) |
  BinOpApp BinOp (Exp a) (Exp a) |
  If (Exp a) (Exp a) (Exp a) |
  Let [(Id, Type, Exp a)] (Exp a)
```
Example: The following code fragment

```ocaml
let int x = 7 in x + 35
```

would be represented like this (before identification):

```
Let [("x", IntType, LitInt 7)]
  (BinOpApp Plus
   (Var "x" ())
   (LitInt 35))
```
Goals of the identification phase:

- Annotate each applied identifier occurrence with attributes of the corresponding variable declaration. I.e., map unannotated AST \( \text{Exp}() \) to annotated AST \( \text{Exp Attr} \).
- Report conflicting variable definitions and undefined variables.

\[
\text{identification} :: \text{Exp}() \rightarrow (\text{Exp Attr}, \text{ErrorMsg})
\]
Example: Before Identification

Let [("x", IntType, LitInt 7)]
(BinOpApp Plus
  (Var "x" ())
  (LitInt 35))
Example: A Compiler Fragment (5)

Example: Before Identification

Let ["x", IntType, LitInt 7)]
(BinOpApp Plus
(Var "x" ()
(LitInt 35))

After identification:

Let ["x", IntType, LitInt 7)]
(BinOpApp Plus
(Var "x" (1, IntType)
(LitInt 35))
enterVar inserts a variable at the given scope level and of the given type into an environment.

- Check that no variable with same name has been defined at the same scope level.
- If not, the new variable is entered, and the resulting environment is returned.
- Otherwise an error message is returned.

\[
\text{enterVar} :: \text{Id} \to \text{Int} \to \text{Type} \to \text{Env} \\
\quad \to \text{Either} \quad \text{Env} \quad \text{ErrorMsg}
\]
Example: A Compiler Fragment (7)

Functions that do the real work:

identAux ::
    Int -> Env -> Exp ()
    -> (Exp Attr, [ErrorMsg])

identDefs ::
    Int -> Env -> [(Id, Type, Exp ())]
    -> [[[Id, Type, Exp Attr]],
         Env,
         [ErrorMsg]]
Example: A Compiler Fragment (8)

\[
\begin{align*}
\text{identDefs} \ l \ \text{env} \ [\] &= ([], \ \text{env}, \ [\]) \\
\text{identDefs} \ l \ \text{env} \ ((i,t,e) : \ ds) &= \\
&= ((i,t,e') : \ ds', \ \text{env''}, \ ms1++ms2++ms3) \\
\text{where} \\
&= (e', \ ms1) = \text{identAux} \ l \ \text{env} \ e \\
&= (\text{env'}, \ ms2) = \\
&\quad \text{case enterVar} \ i \ l \ t \ \text{env} \ of \\
&\quad \quad \text{Left} \ \text{env'} \ \rightarrow (\text{env'}, \ [\]) \\
&\quad \quad \text{Right} \ m \ \rightarrow (\text{env}, \ [m]) \\
&\quad (ds', \ \text{env''}, \ ms3) = \\
&\quad \text{identDefs} \ l \ \text{env'} \ ds
\end{align*}
\]
Error checking and collection of error messages arguably added a lot of *clutter*. The *core* of the algorithm is this:

\[
\begin{align*}
\text{identDefs } l \text{ env } [] &= ([], \text{ env}) \\
\text{identDefs } l \text{ env } ((i,t,e) : ds) &= \\
&\quad ((i,t,e') : ds', \text{ env''}) \\
\text{where} \\
e' &= \text{identAux } l \text{ env } e \\
\text{env'} &= \text{enterVar } i \text{ l t env} \\
(ds', \text{ env''}) &= \text{identDefs } l \text{ env'} ds
\end{align*}
\]

Errors are just a *side effect*. 

Monads bridges the gap: allow effectful programming in a pure setting.
Monads bridges the gap: allow effectful programming in a pure setting.

Key idea: **Computational types**: an object of type $M A$ denotes a *computation* of an object of type $A$. 
Monads bridges the gap: allow effectful programming in a pure setting.

Key idea: **Computational types**: an object of type $MA$ denotes a *computation* of an object of type $A$.

Thus we shall be both pure and impure, whatever takes our fancy!
Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: **Computational types**: an object of type $M^A$ denotes a *computation* of an object of type $A$.
- *Thus we shall be both pure and impure, whatever takes our fancy!*
- Monads originated in Category Theory.
Monads bridges the gap: allow effectful programming in a pure setting.

Key idea: **Computational types**: an object of type $MA$ denotes a *computation* of an object of type $A$.

**Thus we shall be both pure and impure, whatever takes our fancy!**

Monads originated in Category Theory.

Adapted by
- Moggi for structuring denotational semantics
- Wadler for structuring functional programs
Monads

- promote disciplined use of effects since the type reflects which effects can occur;
Monads

- promote disciplined use of effects since the type reflects which effects can occur;
- allow great flexibility in tailoring the effect structure to precise needs;
Answer to Conundrum: Monads (2)

Monads

- promote disciplined use of effects since the type reflects which effects can occur;
- allow great flexibility in tailoring the effect structure to precise needs;
- support changes to the effect structure with minimal impact on the overall program structure;
Monads

- promote disciplined use of effects since the type reflects which effects can occur;
- allow great flexibility in tailoring the effect structure to precise needs;
- support changes to the effect structure with minimal impact on the overall program structure;
- allow integration into a pure setting of real effects such as
  - I/O
  - mutable state.
This Lecture

Pragmatic introduction to monads:

- Effectful computations
- Identifying a common pattern
- Monads as a *design pattern*
Example 1: A Simple Evaluator

data Exp = Lit Integer
  | Add Exp Exp
  | Sub Exp Exp
  | Mul Exp Exp
  | Div Exp Exp

eval :: Exp -> Integer
eval (Lit n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
data Maybe a = Nothing | Just a

safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 + n2)
safeEval \ (\text{Sub\ } e_1\ e_2) =
\begin{align*}
  &\text{case }\text{safeEval } e_1\ \text{of} \\
  &\quad \text{Nothing } \to \text{Nothing} \\
  &\quad \text{Just } n_1 \to \\
  &\quad \quad \text{case }\text{safeEval } e_2\ \text{of} \\
  &\quad\quad \text{Nothing } \to \text{Nothing} \\
  &\quad\quad \text{Just } n_2 \to \text{Just } (n_1 - n_2)
\end{align*}
safeEval (Mul e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 * n2)
safeEval (Div e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 ->
          if n2 == 0
            then Nothing
            else Just (n1 `div` n2)
Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?
Any Common Pattern?

Clearly a lot of code duplication! Can we factor out a common pattern?

We note:

- *Sequencing* of evaluations (or *computations*).
Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- **Sequencing** of evaluations (or computations).
- If one evaluation fails, fail overall.
Clearly a lot of code duplication! Can we factor out a common pattern?

We note:

- **Sequencing** of evaluations (or computations).
- If one evaluation fails, fail overall.
- Otherwise, make result available to following evaluations.
Sequencing Evaluations

evalSeq :: Maybe Integer
      -> (Integer -> Maybe Integer)
      -> Maybe Integer

evalSeq ma f =
  case ma of
    Nothing -> Nothing
    Just a  -> f a
Exercise 1: Refactoring `safeEval`

Rewrite `safeEval`, **case** `Add`, using `evalSeq`:

```haskell
safeEval (Add e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 + n2)

evalSeq ma f =
  case ma of
    Nothing -> Nothing
    Just a  -> f a
```

LiU-FP2010 Part II: Lecture 4 – p.23/52
Exercise 1: Solution

safeEval :: Exp -> Maybe Integer
safeEval (Add e1 e2) =
    evalSeq (safeEval e1)
    (\n1 -> evalSeq (safeEval e2)
        (\n2 -> Just (n1 + n2)))

or

callEval :: Exp -> EvalMonad Integer
safeEval (Add e1 e2) =
safeEval e1 `evalSeq` (\n1 ->
safeEval e2 `evalSeq` (\n2 ->
                    Just (n1 + n2)))
Aside: Scope Rules of $\lambda$-abstractions

The scope rules of $\lambda$-abstractions are such that parentheses can be omitted:

```haskell
safeEval :: Exp -> Maybe Integer
...

safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    Just (n1 + n2)
...
```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    Just (n1 + n2)
safeEval (Sub e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    Just (n1 - n2)
Refactored Safe Evaluator (2)

safeEval (Mul e1 e2) =
  safeEval e1 `evalSeq` \n1 ->
  safeEval e2 `evalSeq` \n2 ->
  Just (n1 * n2)

safeEval (Div e1 e2) =
  safeEval e1 `evalSeq` \n1 ->
  safeEval e2 `evalSeq` \n2 ->
  if n2 == 0
  then Nothing
  else Just (n1 `div` n2)
Inlining \texttt{evalSeq} (1)

\begin{align*}
\text{safeEval} \ (\text{Add} \ e1 \ e2) &= \\
\text{safeEval} \ e1 \ ‘\text{evalSeq}’ \ \backslash n1 \rightarrow \\
\text{safeEval} \ e2 \ ‘\text{evalSeq}’ \ \backslash n2 \rightarrow \\
\text{Just} \ (n1 + n2)
\end{align*}
Inlining `evalSeq` (1)

```haskell
safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` \n1 ->
    safeEval e2 `evalSeq` \n2 ->
    Just (n1 + n2)
```

==

```haskell
safeEval (Add e1 e2) =
    case (safeEval e1) of
        Nothing -> Nothing
        Just a -> (\n1 -> safeEval e2 ...) a
```
Inlining `evalSeq` (2)

\[
\begin{align*}
\text{safeEval} \ (\text{Add} \ e1 \ e2) &= \\
&\text{\hspace{1cm} case} \ (\text{safeEval} \ e1) \ \text{of} \\
&\hspace{1cm} \text{Nothing} \rightarrow \text{Nothing} \\
&\hspace{1cm} \text{Just} \ n1 \rightarrow \text{safeEval} \ e2 \ \text{`evalSeq`} \ (\text{\backslash}n2 \rightarrow \ldots)
\end{align*}
\]
Inlining \texttt{evalSeq} (2)

\begin{verbatim}
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
      Nothing -> Nothing
      Just a -> (\n2 -> ...) a
\end{verbatim}
Inlining evalSeq (3)

=  

safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
      Nothing -> Nothing
      Just n2 -> (Just n1 + n2)

Good exercise: verify the other cases.
• Consider a value of type `Maybe a` as denoting a `computation` of a value of type `a` that `may fail`.
Consider a value of type `Maybe a` as denoting a *computation* of a value of type `a` that *may fail*.

When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
Maybe Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a *computation* of a value of type `a` that *may fail*.

- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.

- I.e. *failure is an effect*, implicitly affecting subsequent computations.
Consider a value of type `Maybe a` as denoting a *computation* of a value of type `a` that *may fail*.

When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.

I.e. *failure is an effect*, implicitly affecting subsequent computations.

Let’s generalize and adopt names reflecting our intentions.
Successful computation of a value:

```haskell
mbReturn :: a -> Maybe a
mbReturn = Just
```

Sequencing of possibly failing computations:

```haskell
mbSeq :: Maybe a -> (a -> Maybe b) -> Maybe b
mbSeq ma f =
  case ma of
    Nothing -> Nothing
    Just a  -> f a
```
Maybe Viewed as a Computation (3)

Failing computation:

```haskell
mbFail :: Maybe a
mbFail = Nothing
```
The Safe Evaluator Revisited

safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = mbReturn n
safeEval (Add e1 e2) =
    safeEval e1 `mbSeq` \n1 ->
    safeEval e2 `mbSeq` \n2 ->
    mbReturn (n1 + n2)

...

safeEval (Div e1 e2) =
    safeEval e1 `mbSeq` \n1 ->
    safeEval e2 `mbSeq` \n2 ->
    if n2 == 0 then mbFail
    else mbReturn (n1 `div` n2))
Example 2: Numbering Trees

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

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
  where

  ntAux :: Tree a -> Int -> (Tree Int,Int)
  ntAux (Leaf _) n = (Leaf n, n+1)
  ntAux (Node t1 t2) n =
    let (t1', n') = ntAux t1 n
    in let (t2', n'') = ntAux t2 n'
       in (Node t1' t2', n'')
Observations

- Repetitive pattern: threading a counter through a sequence of tree numbering computations.
Observations

- Repetitive pattern: threading a counter through a sequence of tree numbering computations.
- It is very easy to pass on the wrong version of the counter!
Observations

- Repetitive pattern: threading a counter through a sequence of tree numbering computations.
- It is very easy to pass on the wrong version of the counter!

Can we do better?
A *stateful computation* consumes a state and returns a result along with a possibly updated state.
Stateful Computations (1)

- A *stateful computation* consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:

```haskell
type S a = Int -> (a, Int)
```

(Only `Int` state for the sake of simplicity.)
A **stateful computation** consumes a state and returns a result along with a possibly updated state.

- The following type synonym captures this idea:
  
  ```
  type S a = Int -> (a, Int)
  ```

  (Only `Int` state for the sake of simplicity.)

- A value (function) of type `S a` can now be viewed as denoting a stateful computation computing a value of type `a`.
When sequencing stateful computations, the resulting state should be passed on to the next computation.
Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.
- I.e. *state updating is an effect*, implicitly affecting subsequent computations. (As we would expect.)
Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a = \text{Int} \rightarrow (a, \text{Int})$):

```haskell
sReturn :: a -> S a
sReturn a = ???
```
Stateful Computations (3)

Computation of a value without changing the state (For ref.: \( S \ a = \text{Int} \rightarrow (a, \text{Int}) \)):

\[
\text{sReturn} :: a \rightarrow S\ a
\]

\[
\text{sReturn} \ a = \lambda n \rightarrow (a, n)
\]
Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\, a = \text{Int} \rightarrow (a, \text{Int})$):

\[
\text{sReturn} :: a \rightarrow S\, a
\]
\[
\text{sReturn}\, a = \lambda n \rightarrow (a, n)
\]

Sequencing of stateful computations:

\[
\text{sSeq} :: S\, a \rightarrow (a \rightarrow S\, b) \rightarrow S\, b
\]
\[
\text{sSeq}\, sa\, f = ???
\]
Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S a = \text{Int} \rightarrow (a, \text{Int})$):

```haskell
sReturn :: a -> S a
sReturn a = \n -> (a, n)
```

Sequencing of stateful computations:

```haskell
sSeq :: S a -> (a -> S b) -> S b
sSeq sa f = \n ->
    let (a', n') = sa n
    in f a n'
```
Stateful Computations (4)

Reading and incrementing the state
(For ref.: $S \ a = \text{Int} \rightarrow (a, \text{Int})$):

```haskell
sInc :: S Int
sInc = \n -> (n, n + 1)
```
Numbering trees revisited

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

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)

where

ntAux :: Tree a -> S (Tree Int)
ntAux (Leaf _) =
    sInc `sSeq` \n -> sReturn (Leaf n)
ntAux (Node t1 t2) =
    ntAux t1 `sSeq` \t1' ->
    ntAux t2 `sSeq` \t2' ->
    sReturn (Node t1' t2')
Observations

- The “plumbing” has been captured by the abstractions.
Observations

- The “plumbing” has been captured by the abstractions.
- In particular:
  - counter no longer manipulated directly
  - no longer any risk of “passing on” the wrong version of the counter!
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
  - A function constructing a computation by sequencing computations
Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
  - A function constructing a computation by sequencing computations
- In fact, both examples are instances of the general notion of a *MONAD*. 

**•**
Monads in Functional Programming

A monad is represented by:

- A type constructor
  \[ M :: * \rightarrow * \]
  \( M \ M T \) represents computations of a value of type \( T \).

- A polymorphic function
  \[ \text{return} :: a \rightarrow M a \]
  for lifting a value to a computation.

- A polymorphic function
  \[ (\gg\gg=) :: M a \rightarrow (a \rightarrow M b) \rightarrow M b \]
  for sequencing computations.
Exercise 2: `join` and `fmap`

Equivalently, the notion of a monad can be captured through the following functions:

```
return :: a -> M a
join :: (M (M a)) -> M a
fmap :: (a -> b) -> (M a -> M b)
```

`join` “flattens” a computation, `fmap` “lifts” a function to map computations to computations.

Define `join` and `fmap` in terms of `>>= (and return)`, and `>>= in terms of join and fmap`.

```
(>>=) :: M a -> (a -> M b) -> M b
```
Exercise 2: Solution

join :: M (M a) → M a
join mm = mm >>= id

fmap :: (a → b) → M a → M b
fmap f m = m >>= \a → return (f a)

or:

fmap :: (a → b) → M a → M b
fmap f m = m >>= return . f

(>>>=) :: M a → (a → M b) → M b
m >>= f = join (fmap f m)
Additionally, the following \textit{laws} must be satisfied:

\begin{align*}
\text{return } x \implies f &= f x \\
m \implies \text{return} &= m \\
(m \implies f) \implies g &= m \implies (\lambda x \rightarrow f x \implies g)
\end{align*}

I.e., \text{return} is the right and left identity for \implies, and \implies is associative.
Exercise 3: The Identity Monad

The *Identity Monad* can be understood as representing *effect-free* computations:

```haskell
type I a = a
```

1. Provide suitable definitions of `return` and `>>=`

2. Verify that the monad laws hold for your definitions.
Exercise 3: Solution

\[ \text{return} :: a \rightarrow \text{I} \ a \]
\[ \text{return} = \text{id} \]

\[ (\gg=) :: \text{I} \ a \rightarrow (a \rightarrow \text{I} \ b) \rightarrow \text{I} \ b \]
\[ m \gg= f = f \ m \]
\[ \text{or: } (\gg=) = \text{flip} \ ($) \]

Simple calculations verify the laws, e.g.:

\[ \text{return} \ x \gg= f = \text{id} \ x \gg= f = x \gg= f = f \ x \]
The notion of a monad originated in Category Theory. There are several equivalent definitions (Benton, Hughes, Moggi 2000):

- **Kleisli triple/triple in extension form**: Most closely related to the >>= version:

  A **Kleisli triple** over a category $\mathcal{C}$ is a triple $(T, \eta, _*)$, where $T : |\mathcal{C}| \rightarrow |\mathcal{C}|$, $\eta_A : A \rightarrow TA$ for $A \in |\mathcal{C}|$, $f^* : TA \rightarrow TB$ for $f : A \rightarrow TB$.

  (Additionally, some laws must be satisfied.)
Monads in Category Theory (2)

- **Monad/triple in monoid form:** More akin to the `join/fmap` version:

  A **monad** over a category \( \mathcal{C} \) is a triple \((T, \eta, \mu)\), where \( T : \mathcal{C} \to \mathcal{C} \) is a functor, \( \eta : \text{id}_{\mathcal{C}} \to T \) and \( \mu : T^2 \to T \) are natural transformations.

  (Additionally, some commuting diagrams must be satisfied.)
Reading


• *All About Monads.*
  
  [http://www.haskell.org/all_about_monads](http://www.haskell.org/all_about_monads)