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)

• 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; }
}
```

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

• `x` refers to the **instance variable** `x`
• `n` refers to the **argument** `n`. 
Example: A Compiler Fragment (2)

Consider an AST $\text{Exp}$ for a simple expression language. $\text{Exp}$ is a parameterized type: the \textbf{type parameter} $a$ allows variables to be annotated with an attribute of type $a$.

\begin{verbatim}
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)
\end{verbatim}

Example: A Compiler Fragment (3)

Example: The following code fragment

\begin{verbatim}
let int x = 7 in x + 35
\end{verbatim}

would be represented like this (before identification):

\begin{verbatim}
Let ["x", IntType, LitInt 7] (BinOpApp Plus (Var "x" ()) (LitInt 35))
\end{verbatim}

Example: A Compiler Fragment (4)

Goals of the \textbf{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.

\begin{verbatim}
identification :: Exp () -> (Exp Attr, [ErrorMsg])
\end{verbatim}

Example: A Compiler Fragment (5)

Example: Before Identification

\begin{verbatim}
Let ["x", IntType, LitInt 7] (BinOpApp Plus
 (Var "x" ())
 (LitInt 35))
\end{verbatim}

After identification:

\begin{verbatim}
Let ["x", IntType, LitInt 7] (BinOpApp Plus
 (Var "x" (1, IntType))
 (LitInt 35))
\end{verbatim}
Example: A Compiler Fragment (6)

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.

```haskell
enterVar :: Id -> Int -> Type -> Env
          -> Either Env ErrorMsg
```

Example: A Compiler Fragment (7)

Functions that do the real work:

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

identDefs ::
          Int -> Env -> [[(Id, Type, Exp ())]]
          -> ([(Id, Type, Exp Attr)],
              Env,
              [ErrorMsg])
```

Example: A Compiler Fragment (8)

```haskell
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) = ((i,t,e') : ds', env'', msl++ms2++ms3)
where
  (e', msl) = identAux l env e
  (env', ms2) = case enterVar i l t env of
                 Left env' -> (env', [])
                 Right m -> (env, [m])
  (ds', env'', ms3) = identDefs l env' ds
```

Example: A Compiler Fragment (9)

Error checking and collection of error messages arguably added a lot of clutter. The core of the algorithm is this:

```haskell
identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) = ((i,t,e') : ds', env'')
where
e' = identAux l env e
e'' = enterVar i l t env (ds', env'') = identDefs l env' ds
```

Errors are just a side effect.
**Answer to Conundrum: Monads (1)**

- Monads bridge 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

**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;
- 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
```
Making the Evaluator Safe (1)

data Maybe a = Nothing | Just a

safeEval :: Exp -> Maybe Integer

safeEval (Lit n) = Just n

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

Making the Evaluator Safe (2)

safeEval (Sub el e2) =
    case safeEval el of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 - n2)

Making the Evaluator Safe (3)

safeEval (Mul el e2) =
    case safeEval el of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 * n2)

Making the Evaluator Safe (4)

safeEval (Div el e2) =
    case safeEval el 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?

We note:

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

Sequencing Evaluations

```haskell
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)
```

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

Exercise 1: Solution

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

Or

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

```haskell
safeEval (Add e1 e2) = evalSeq (safeEval e1)
                             (\n1 -> evalSeq (safeEval e2)
                             (\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)
...
```

Refactored Safe Evaluator (1)

```haskell
safeEval :: Exp -> Maybe IntegersafeEval (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)

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 `evalSeq` (1)

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

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

safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
      Nothing -> Nothing
      Just a -> (\n2 -> ...) a

Inlining evalSeq (3)

safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
      Nothing -> Nothing
      Just a -> (\n2 -> ...) a

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.
- Let’s generalize and adopt names reflecting our intentions.

Maybe Viewed as a Computation (2)

Successful computation of a value:

mbReturn :: a -> Maybe a
mbReturn = Just

Sequencing of possibly failing computations:

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

```haskell
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

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

numberTree :: Tree a -> Tree Int
numberTree t = \n1 ->
    numberTree t1 `mbSeq` \n2 ->
    numberTree t2 `mbSeq` (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.

- It is very easy to pass on the wrong version of the counter!

Can we do better?
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:
  
  ```
  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 \( \text{a} \).

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.: \( S \ a = \text{Int} \rightarrow (\text{a}, \text{Int}) \)):

```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.: \( S \ a = \text{Int} \rightarrow (\text{a}, \text{Int}) \)):

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

```haskell
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.
- 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.
- 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
  ```haskell
  M :: * -> *
  M T represents computations of a value of type T.
  ```
- A polymorphic function
  ```haskell
  return :: a -> M a
  ```
  for lifting a value to a computation.
- A polymorphic function
  ```haskell
  (>>=) :: M a -> (a -> M b) -> M b
  ```
  for sequencing computations.
**Exercise 2: join and fmap**

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

\[
\begin{align*}
\text{return} & : a \to M a \\
\text{join} & : (M (M a)) \to M a \\
\text{fmap} & : (a \to b) \to (M a \to M b)
\end{align*}
\]

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.

\[
(\gg\gg) : M a \to (a \to M b) \to M b
\]

**Exercise 2: Solution**

\[
\begin{align*}
\text{join} & : M (M a) \to M a \\
\text{join} \ mm = mm \gg\gg id \\
\text{fmap} & : (a \to b) \to M a \to M b \\
\text{fmap} \ f \ m = m \gg\gg \ \lambda x \to \text{return} \ (f \ x) \\
\text{Or:} \\
\text{fmap} & : (a \to b) \to M a \to M b \\
\text{fmap} \ f \ m = m \gg\gg \text{return} \ . \ f
\end{align*}
\]

\[
(\gg\gg) : M a \to (a \to M b) \to M b \\
\m \gg\gg f = \text{join} \ (\text{fmap} \ f \ m)
\]

**Monad laws**

Additionally, the following **laws** must be satisfied:

\[
\begin{align*}
\text{return} \ x \gg\gg f & = f \ x \\
\m \gg\gg \text{return} & = \m \\
(m \gg\gg f) \gg\gg g & = m \gg\gg (\lambda x \to f \ x \gg\gg g)
\end{align*}
\]

I.e., return is the right and left identity for >>=, and >>= is associative.

**Exercise 3: The Identity Monad**

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

\[
\text{type I a} = a
\]

1. Provide suitable definitions of return and >>=.
2. Verify that the monad laws hold for your definitions.
Exercise 3: Solution

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

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

Simple calculations verify the laws, e.g.:

\[
\text{return} \ x \text{ >>=} f = \text{id} \ x \text{ >>=} f \\
= \ x \text{ >>=} f \\
= f \ x
\]

Monads in Category Theory (1)

The notion of a monad originated in Category Theory. There are several equivalent definitions (Benton, Hughes, Moggi 2000):

- **Klesli triple/triple in extension form:** Most closely related to the \(>>=\) version:
  A *Klesli triple* over a category \(\mathcal{C}\) is a triple \((\text{T}, \eta, \_\_\_\_\_\_\_*\_\_\_\_\_\_\_\_\_\_)\), where \(\text{T} : |\mathcal{C}| \rightarrow |\mathcal{C}|\), \(\eta_A : A \rightarrow \text{T} A\) for \(A \in |\mathcal{C}|\), \(f^* : \text{T} A \rightarrow \text{T} B\) for \(f : A \rightarrow B\).
  (Additionally, some laws must be satisfied.)

Monads in Category Theory (2)

- **Monad/triple in monoid form:** More akin to the \text{join/fmap} version:
  A *monad* over a category \(\mathcal{C}\) is a triple \((\text{T}, \eta, \mu)\), where \(\text{T} : \mathcal{C} \rightarrow \mathcal{C}\) is a functor, \(\eta : \text{id}_{\mathcal{C}} \rightarrow \text{T}\) and \(\mu : \text{T}^2 \rightarrow \text{T}\) are natural transformations.
  (Additionally, some commuting diagrams must be satisfied.)

Reading

- *All About Monads.*
  http://www.haskell.org/all_about_monads