**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.
- Disciplined use of 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`.
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 \( a \).

\[
\text{data } \text{Exp} a \ = \ \text{LitInt } \text{Int} | \ \text{Var } \text{Id } a |
\text{UnOpApp } \text{UnOp } (\text{Exp } a) |
\text{BinOpApp } \text{BinOp } (\text{Exp } a) (\text{Exp } a) |
\text{If } (\text{Exp } a) (\text{Exp } a) (\text{Exp } a) |
\text{Let } [(\text{Id}, \text{Type}, \text{Exp } a)] (\text{Exp } a)
\]

Goals of the \textit{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 \([(\text{"x"}, \text{IntType}, \text{LitInt 7})] \)
\( \text{Var } \text{"x" } () \)
\( \text{LitInt 35} \)

After identification:

Let \([(\text{"x"}, \text{IntType}, \text{LitInt 7})] \)
\( \text{Var } \text{"x" } (1, \text{IntType}) \)
\( \text{LitInt 35} \)
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.

enterVar :: Id -> Int -> Type -> Env -> Either Env 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)

identDefs 1 env [] = ([], env, [])
identDefs 1 env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
  (e', ms1) = identAux 1 env e
  (env'', ms2) =
    case enterVar i l t env of
      Left env' -> (env', [])
      Right m -> (env, [m])
  (ds', env''', ms3) =
    identDefs 1 env' ds

Example: A Compiler Fragment (9)

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

identDefs 1 env [] = ([], env)
identDefs 1 env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'')
  where
  e' = identAux 1 env e
  env' = enterVar i l t env
  (ds', env''') = identDefs 1 env' ds

Errors are just a side effect.
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 originated 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;
- 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.

Example 1: A Simple Evaluator

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

eval :: Exp -> Int
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 (Sub e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 - n2)

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?

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)

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)))
```
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 Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
  safeEval e1 'evalSeq' \n1 ->
  safeEval e2 'evalSeq' \n2 ->
  Just (n1 + n2)
...
```

Refactored Safe Evaluator (2)

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

\[
\text{safeEval} (\text{Add} \ e_1 \ e_2) = \\
\quad \text{case} \ (\text{safeEval} \ e_1) \ of \\
\quad \quad \text{Nothing} \to \text{Nothing} \\
\quad \quad \text{Just} \ n_1 \to \text{safeEval} \ e_2 \ 'evalSeq' \ (\ \ n_2 \to \ldots)
\]

Inlining `evalSeq` (3)

\[
\text{safeEval} (\text{Add} \ e_1 \ e_2) = \\
\quad \text{case} \ (\text{safeEval} \ e_1) \ of \\
\quad \quad \text{Nothing} \to \text{Nothing} \\
\quad \quad \text{Just} \ n_1 \to \text{case} \ \text{safeEval} \ e_2 \ of \\
\quad \quad \quad \text{Nothing} \to \text{Nothing} \\
\quad \quad \quad \text{Just} \ a \to \ (\ \ n_2 \to \ldots) \ a
\]

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:

\[
\text{mbReturn} :: a \to \text{Maybe } a \\
\text{mbReturn} = \text{Just}
\]

Sequencing of possibly failing computations:

\[
\text{mbSeq} :: \text{Maybe } a \to (a \to \text{Maybe } b) \to \text{Maybe } b \\
\text{mbSeq} \text{ ma f} = \\
\quad \text{case} \ \text{ma} \ of \\
\quad \quad \text{Nothing} \to \text{Nothing} \\
\quad \quad \text{Just} \ a \to \ f \ a
\]
Maybe Viewed as a Computation (3)

Failing computation:

\[
\begin{align*}
mbFail :: \text{Maybe } a \\
mbFail = \text{Nothing}
\end{align*}
\]

The Safe Evaluator Revisited

\[
\begin{align*}
\text{safeEval} :: \text{Exp} \to \text{Maybe Integer} \\
\text{safeEval} (\text{Lit } n) &= \text{mbReturn } n \\
\text{safeEval} (\text{Add } e1 \ e2) &= \text{safeEval } e1 \text{ mbSeq' } n1 \to \\
&\quad \text{safeEval } e2 \text{ mbSeq' } n2 \to \\
&\quad \text{mbReturn } (n1 + n2) \\
\ldots
\end{align*}
\]

Example 2: Numbering Trees

\[
\begin{align*}
\text{data Tree } a &= \text{Leaf } a \mid \text{Node } (\text{Tree } a) (\text{Tree } a) \\
\text{numberTree} :: \text{Tree } a \to \text{Tree Int} \\
\text{numberTree } t &= \text{fst } (\text{ntAux } t \ 0) \\
\text{where}
\end{align*}
\]

\[
\begin{align*}
\text{ntAux} :: \text{Tree } a \to \text{Int } \to \text{(Tree Int,Int)} \\
\text{ntAux } (\text{Leaf } _) &= (\text{Leaf } n, \ n+1) \\
\text{ntAux } (\text{Node } t1 \ t2) n &= \\
&\quad \text{let } (t1', \ n') = \text{ntAux } t1 \ n \\
&\quad \text{in } \quad \text{let } (t2', \ n'') = \text{ntAux } t2 \ n' \\
&\quad \text{in } \quad \text{(Node } t1' \ t2', \ n'')
\end{align*}
\]

Observations

- Repetitive pattern: threading a counter through a \textbf{sequence} of tree numbering \textbf{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:
  ```haskell
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`.

**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 = Int -> (a, 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 = Int -> (a, 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
  
  \[ M :: \ast \rightarrow \ast \]

  \( M T \) represents computations of a value of type \( T \).
- A polymorphic function
  
  \[ return :: a \rightarrow M a \]

  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:

\[\text{return} :: a \rightarrow M a\]
\[\text{join} :: (M (M a)) \rightarrow M a\]
\[\text{fmap} :: (a \rightarrow b) \rightarrow (M a \rightarrow M b)\]

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.

\[(\ggg) :: M a \rightarrow (a \rightarrow M b) \rightarrow M b\]

Exercise 2: Solution

\[\text{join} :: M (M a) \rightarrow M a\]
\[\text{join} \; mm = mm \ggg \text{id}\]

\[\text{fmap} :: (a \rightarrow b) \rightarrow M a \rightarrow M b\]
\[\text{fmap} \; f \; m = m \ggg \lambda a \rightarrow \text{return} \; (f \; a)\]

Or:

\[\text{fmap} :: (a \rightarrow b) \rightarrow M a \rightarrow M b\]
\[\text{fmap} \; f \; m = m \ggg \text{return} \; \cdot \; f\]

\[(\ggg) :: M a \rightarrow (a \rightarrow M b) \rightarrow M b\]
\[m \ggg f = \text{join} \; (\text{fmap} \; f \; m)\]

Monad laws

Additionally, the following laws must be satisfied:

\[\text{return} \; x \ggg f = f \; x\]
\[m \ggg \text{return} = m\]
\[(m \ggg f) \ggg g = m \ggg (\lambda x \rightarrow f \; x \ggg g)\]

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

Monads in Category Theory (1)

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 \((T, \eta, \_\star)\), where \(T : |\mathcal{C}| \rightarrow |\mathcal{C}|\),
  \[
  \eta_A : A \rightarrow TA \quad \text{for} \quad A \in |\mathcal{C}|,
  \]
  \[
  f^\star : TA \rightarrow TB
  \]
  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 \( C \) is a triple \((T, \eta, \mu)\), where \( T : C \to C \) is a functor,\n  
  \( \eta : \text{id}_C \to T \) and \( \mu : T^2 \to T \) are natural transformations.

  (Additionally, some commuting diagrams must be satisfied.)

Reading


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