

# MGS 2012: FUN Lecture 3

## Monads

Henrik Nilsson

University of Nottingham, UK

MGS 2012: FUN Lecture 3 – p.1/50

## 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

MGS 2012: FUN Lecture 3 – p.2/50

## 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.

MGS 2012: FUN Lecture 3 – p.3/50

## Example: A Compiler Fragment (1)

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

```
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`.

MGS 2012: FUN Lecture 3 – p.4/50

## Example: A Compiler Fragment (2)

Consider an AST `Exp` for a simple expression language. `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)
```

MGS 2012: FUN Lecture 3 – p.5/50

## Example: A Compiler Fragment (3)

Example: The following code fragment

```
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))
```

MGS 2012: FUN Lecture 3 – p.6/50

## Example: A Compiler Fragment (4)

Goals of the **identification** phase:

- Annotate each applied identifier occurrence with attributes of the corresponding variable declaration.

I.e., map unannotated AST `Exp ()` to annotated AST `Exp Attr`.

- Report conflicting variable definitions and undefined variables.

identification ::

```
Exp () -> Exp Attr [ErrorMsg]
```

MGS 2012: FUN Lecture 3 – p.7/50

## 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))
```

MGS 2012: FUN Lecture 3 – p.8/50

## 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 ErrorMessage
```

MGS 2012: FUN Lecture 3 – p.9/50

## 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]
```

MGS 2012: FUN Lecture 3 – p.10/50

## Example: A Compiler Fragment (8)

```
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
    (e', ms1) = 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
```

MGS 2012: FUN Lecture 3 – p.11/50

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

Errors are just a **side effect**.

MGS 2012: FUN Lecture 3 – p.12/50

## Answer to Conundrum: Monads (1)

- 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

MGS 2012: FUN Lecture 3 – p.13/50

## 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.

MGS 2012: FUN Lecture 3 – p.14/50

## This Lecture

Pragmatic introduction to monads:

- Effectful computations
- Identifying a common pattern
- Monads as a **design pattern**

MGS 2012: FUN Lecture 3 – p.15/50

## 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
```

MGS 2012: FUN Lecture 3 – p.16/50

## Making the Evaluator Safe (1)

```
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)
```

MGS 2012: FUN Lecture 3 – p.17/50

## Making the Evaluator Safe (3)

```
safeEval (Mul e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 * n2)
```

MGS 2012: FUN Lecture 3 – p.19/50

## Making the Evaluator Safe (2)

```
safeEval (Sub e1 e2) =
  case safeEval e1 of
    Nothing -> Nothing
    Just n1 ->
      case safeEval e2 of
        Nothing -> Nothing
        Just n2 -> Just (n1 - n2)
```

MGS 2012: FUN Lecture 3 – p.18/50

## Making the Evaluator Safe (4)

```
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)
```

MGS 2012: FUN Lecture 3 – p.20/50

## 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.

MGS 2012: FUN Lecture 3 – p.21/50

## Sequencing Evaluations

```
evalSeq :: Maybe Integer
         -> (Integer -> Maybe Integer)
         -> Maybe Integer
evalSeq ma f =
  case ma of
    Nothing -> Nothing
    Just a   -> f a
```

MGS 2012: FUN Lecture 3 – p.22/50

## Exercise 1: Refactoring safeEval

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

```
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
```

MGS 2012: FUN Lecture 3 – p.23/50

## Exercise 1: Solution

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

Or

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

MGS 2012: FUN Lecture 3 – p.24/50

## Aside: Scope Rules of $\lambda$ -abstractions

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

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

MGS 2012: FUN Lecture 3 – p.25/50

## Refactored Safe Evaluator (1)

```
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)
```

MGS 2012: FUN Lecture 3 – p.26/50

## 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)
```

MGS 2012: FUN Lecture 3 – p.27/50

## Inlining evalSeq (1)

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

MGS 2012: FUN Lecture 3 – p.28/50

## Inlining evalSeq (2)

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

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

MGS 2012: FUN Lecture 3 – p.29/50

## 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.

MGS 2012: FUN Lecture 3 – p.30/50

## 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.

MGS 2012: FUN Lecture 3 – p.31/50

## 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
```

MGS 2012: FUN Lecture 3 – p.32/50



## Maybe Viewed as a Computation (3)

Failing computation:

```
mbFail :: Maybe a
mbFail = Nothing
```

MGS 2012: FUN Lecture 3 – p.33/50

## 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))
```

MGS 2012: FUN Lecture 3 – p.34/50

## 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'')
```

MGS 2012: FUN Lecture 3 – p.35/50

## 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?

MGS 2012: FUN Lecture 3 – p.36/50

## 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  $a$ .

MGS 2012: FUN Lecture 3 – p.37/50

## Stateful Computations (3)

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

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

Sequencing of stateful computations:

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

MGS 2012: FUN Lecture 3 – p.39/50

## 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.)

MGS 2012: FUN Lecture 3 – p.38/50

## Stateful Computations (4)

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

```
sInc :: S Int
sInc = \n -> (n, n + 1)
```

MGS 2012: FUN Lecture 3 – p.40/50

## 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')
```

MGS 2012: FUN Lecture 3 – p.41/50

## 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!

MGS 2012: FUN Lecture 3 – p.42/50

## 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**.

MGS 2012: FUN Lecture 3 – p.43/50

## Monads in Functional Programming

A monad is represented by:

- A type constructor

```
M :: * -> *
```

M T represents computations of a value of type T.

- A polymorphic function

```
return :: a -> M a
```

for lifting a value to a computation.

- A polymorphic function

```
(>>=) :: M a -> (a -> M b) -> M b
```

for sequencing computations.

MGS 2012: FUN Lecture 3 – p.44/50

## 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
```

MGS 2012: FUN Lecture 3 – p.45/50

## 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)
```

MGS 2012: FUN Lecture 3 – p.46/50

## Monad laws

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

```
return x >>= f = f x
m >>= return = m
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

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

MGS 2012: FUN Lecture 3 – p.47/50

## 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  $(T, \eta, \_*)$ , 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.)

MGS 2012: FUN Lecture 3 – p.48/50

## 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  $T : \mathcal{C} \rightarrow \mathcal{C}$  is a functor,  $\eta : \text{id}_{\mathcal{C}} \rightarrow T$  and  $\mu : T^2 \rightarrow T$  are natural transformations.

(Additionally, some commuting diagrams must be satisfied.)

MGS 2012: FUN Lecture 3 – p.49/50

## Reading

- Philip Wadler. The Essence of Functional Programming. *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, 1992.
- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- *All About Monads*.  
[http://www.haskell.org/haskellwiki/all\\_about\\_monads](http://www.haskell.org/haskellwiki/all_about_monads)

MGS 2012: FUN Lecture 3 – p.50/50