Tutorial: Monads for the Working Lisp Programmer

Ravi Nanavati and Jeff Newbern

ILC 2009

March 22, 2009

Tutorial Outline

- Introduction
- Tutorial Contents

  I. Monads in Haskell

  II. Translating Monads to Lisp

  III. Clojure Monad Library

  IV. Our Library Extensions

  V. Interpreter Example

Please interact and ask questions!

Tutorial Exercises

- We want this to be hands-on
  - We have 5 different exercises
  - Our goal is to leave you with new ideas and concrete experience that you can apply to future projects
- Software requirements:
  - PLT Scheme (or another Lisp if you're OK translating on the fly) [3 exercises]
  - The latest stable Clojure (20090320) and clojure-contrib releases [2 exercises]

Tutorial Exercises

I. Monads in Haskell

II. Translating Monads to Lisp

- **Exercise: Translating a monad**
- **Exercise: Implementing custom monadic syntax**
- **Exercise: Ambiguous parsing with a list monad**

III. Clojure Monad Library

IV. Our Library Extensions

V. Interpreter Example

Tutorial Exercises

I. Monads in Haskell

II. Translating Monads to Lisp

III. Clojure Monad Library

- **Exercise: Implementing mapm**

IV. Our Library Extensions

V. Interpreter Example

Tutorial Exercises

I. Monads in Haskell

II. Translating Monads to Lisp

III. Clojure Monad Library

IV. Our Library Extensions

V. Interpreter Example

- **Exercise: Building a modular language fragment**

Tutorial Online

- http://github.com/jnewbern/monad-tutorial/tree/new-master/
  - exercises, solutions and slides subdirectories
  - slides/MonadTutorial.pdf is this presentation
  - there might be updates after today (time permitting, no promises)
  - most notably, compatibility tweaks for newer versions of Clojure and its monad library
- Contact us
  - Ravi: ravi_n@alum.mit.edu
  - Jeff: jnewbern@yahoo.com

Monad

A design pattern for composable effectful computations

Extends values with structure to model effects.
t becomes t_{effect} when t is in the "effect" monad.

Allows effects to be combined when working with values in the monad. Like function application + effect accumulation:
t_{effect} -> (t -> t'_{effect}) -> t'_{effect}

Abstraction layer for the effect machinery so that code is cleaner and effects can be controlled separately from computation.

Monads in Haskell

```
class Monad m where
    -- Sequentially compose two actions, passing
    -- any value produced by the first as an
    -- argument to the second.
    (>>=) :: m a -> (a -> m b) -> m b
    -- Inject a value into the monadic type.
    return :: a -> m a
```

Laws:
left-identity: (return x) >>= f == f x
right-identity: mv >>= return == mv
associativity: (mv >>= f) >>= g == mv >>= (\x -> f x >> g)

Effect: Partiality

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return x = Just x
    Nothing >>= _ = Nothing
    Just x  >>= f = f x
```

Example: Partiality

```
divide n d | d != 0 = return (n/d)
           | d == 0 = Nothing

-- compute a / (b / c)
myfun a b c =
  (divide b c) >>= (\d -> divide a d)

-- sample execution
myfun 30 14 7 ==> Just 15
myfun 30  0 7 ==> Nothing
myfun 30 14 0 ==> Nothing
```

Effect: Failure

```
data Either a b = Left a | Right b

instance Monad (Either e) where
    return x = Right x
    Left err >>= _ = Left err
    Right x  >>= f = f x

fail err = Left err
```

Example: Failure

```
lookup_by_name n db =
  case (find n db) of
    (Just entry) -> return entry
    Nothing      -> fail “name not found”

get_email_addr entry =
  if (has_email entry)
  then return (email_addr entry)
  else fail “no email address”

name_to_email name db =
  (lookup_by_name name db) >>= get_email_addr
```

Effect: State

```
newtype State st a =
   State { runState :: st -> (a,st) }

instance Monad (State st) where
    return x = State $ \s -> (x,s)
    mv >>= f  = State $ \s ->
      let (x, s') = runState mv s
      in runState (f x) s'

get = State $ \s -> (s,s)
put s = State $ \_ -> ((),s)
```

Example: State

```
-- build a histogram of arguments to a function
use_and_record fn x =
  get >>= \hist ->
  put (incr hist x) >>= \_ ->
  return (fn x)

-- with “do notation”
use_and_record fn x =
  do hist ← get
     let new_hist = incr hist x
     put new_hist
     return (fn x)
```

The Monad Design Pattern

```
-- define data or newtype for modeling the effect
???


instance Monad ??? where
    -- add effect structure to pure value
    return x = ???
    -- apply f to value from mv and
    -- combine effects of mv and f's result
    mv >>= f  = ???


-- monad-specific functions for injecting and
-- working with effects
???
```

Monad Functions

```
-- sequentially compose effects, discarding the
-- value produced by the first argument
>> :: (Monad m) => m a -> m b -> mb
mv1 >> mv2 = mv1 >>= (\_ -> mv2)

-- lift a function on values into a monad
liftM :: (Monad m) -> (a -> b) -> (m a -> m b)
liftM f mv = do { x1 ← mv; return (f x1) }

-- evaluate each action in sequence from left
-- to right, collecting the results
sequence :: (Monad m) => [m a] -> m [a]
sequence ms = foldr k (return []) ms
  where k mv mv' =
    do { x <- mv; xs <- mv'; return (x:xs) }
```

many more: mapM, filterM, foldM, replicateM, liftM2, liftM3, ...

Monad Transformers

Often you want to use multiple effects in combination, e.g. state and errors.

A monad transformer adds an effect to a base monad.

Multiple monad transformers can be used to layer on multiple effects. The combination of transformers and the base monad is called a "transformer stack".

StateT

```
newtype StateT st m a =
    StateT { runStateT :: st -> m (a,st) }

instance (Monad m) => Monad (StateT st m) where
    return x = StateT $ \s -> return (x, s)
    mv >>= f  = StateT $ \s -> do
        (x, s') <- runStateT mv s
        runStateT (f x) s'

getT = StateT $ \s -> return (s,s)
putT s = StateT $ \_ -> return ((),s)
```

ErrorT

```
newtype ErrorT e m a =
   ErrorT { runErrorT :: m (Either e a) }


instance (Monad m) => Monad (ErrorT e m) where
    return x = ErrorT $ return (Right x)
    mv >>= f  = ErrorT $ do
        a <- runErrorT mv
        case a of
            Left  err -> return (Left err)
            Right x   -> runErrorT (f x)


failT err = ErrorT $ return (Left err)
```

Working with a Transformer Stack

Order matters!

Contrast `ErrorT e (State st)` vs. `StateT st (Error e)`.

Working with a Transformer Stack

Order matters!

Contrast `ErrorT e (State st)` vs. `StateT st (Error e)`.

`ErrorT e (State st)` ==> `st -> (Either e a, st)`

Failure does not produce a value, but still gives a valid state!

Working with a Transformer Stack

Order matters!

Contrast `ErrorT e (State st)` vs. `StateT st (Error e)`.

`ErrorT e (State st)` ==> `st -> (Either e a, st)`

Failure does not produce a value, but still gives a valid state!

`StateT st (Error e)` ==> `st -> Either e (a, st)`

Failure does not produce a value or a valid state!

Lifting Operations

When working with transformer stacks it is often required to lift values or operations in a lower monad into the combined monad, or to recover a value in a lower monad from a value in the combined monad.

In Haskell, there is a type class operation for monad transformers to lift a value from the lower monad:

```
lift :: (Monad m) => m a -> t m a
```

To lift through multiple levels, the operation can be composed multiple times (e.g., `lift . lift . lift`).

To move down the stack the various run functions are used to “peel the onion” back to the desired monad level.

How do you make a monad outside of Haskell?

- Two core functions
  - bind
  - return
- Implementations of effectful operations
  - failure for an error monad
  - get and put for a state monad
  - callcc for a continuation monad
  - whatever else you want
- The rest is mainly syntax, higher-order functions and polymorphism

Monads and Lisp fit well

- Monadic syntax can be first-class (macros)
  - Better than Haskell – where do-notation is not first-class
- Lisp has higher-order functions and closures
  - Required for bind, monadic map and many other monad utility functions
- Dynamic types provide polymorphism
  - Many monadic functions are also polymorphic (e.g. bind, return)
  - Tradeoff: mixing effects can be trickier

Example: Environment Monad

- Effect: computations that implicitly thread some read-only information

```
(define (return v) (lambda (env) v))
(define (bind f mv)
   (lambda (env) (f (mv env) env)))
```

- In this bind function, the same environment is used twice
  - Compare this to bind for a state monad

Environment Monad Operations

```
; access the local environment
(define capture-env
    (lambda (env) env)

; change the environment for a
; sub-computation
(define (local-env f mv)
    (lambda (env)
        (mv (f env))))
```

Exercise: List Monad

- Represents ambiguous computations with varying numbers of results
- `(return v)` has exactly one result
- `m-zero` means no results
- `(m-plus a b c ...)` joins together the possible results of a and b
- as always, `bind` handles the sequencing

- How would you write these functions?

Exercise: List Monad (continued)

- Sample Scheme code is available at: monad-tutorial/exercises/SchemeMonads/List.ss
- Exercise goals:
  - Building a monad implementation in a familiar language
  - Understanding direct monadic programming with bind and return

Exercise: Monadic Syntax

- Direct programming with bind and return isn't terribly convenient
- Haskell has do-notation to deal with this
- Lisp macros can be used to make more convenient syntax
- Example: `letM` and `letM*` macros
- Exercise instructions at: monad-tutorial/exercises/SchemeMonads/Monad.ss
- Goal: Understanding how to build useful monadic syntax in Lisp

Putting this all together: A monadic evaluator fragment

```
(define (analyze-function name body)
  (let ((body-code (analyze body)))
  (letM ((env capture-env))
  (return
    (lambda (val)
      ((with-binding name val body)
        env))))))
```

- monad-tutorial/scheme/Env.ss and EnvInterp.ss has the full code

Exercise: Monadic String Parsing

- Uses List monad from first exercise
- Parse a string into a word or a number (decimal or hexadecimal)
- Scheme starter code in:
  monad-tutorial/exercises/SchemeMonads/ParseString.ss
- Goal: Write an interesting monadic program using monadic syntax

Monads in Clojure

Clojure has a clojure.contrib.monads library, written by Konrad Hinsen.

It is patterned off of Haskell's monad library, but uses macros and structures instead of Haskell's type classes. In some ways this is nicer than Haskell's monad support!

This library is growing and undergoes frequent revisions, so it is worth updating it frequently to get the latest goodies!

Clojure Monad Structure

```
; create an anonymous monad structure
(defmacro monad
   [operations]
   `(let [~'m-bind   ::undefined
          ~'m-result ::undefined
          ~'m-zero   ::undefined
          ~'m-plus   ::undefined
          ~@operations]
      {:m-result ~'m-result
       :m-bind ~'m-bind
       ; these are optional
       :m-zero ~'m-zero
       :m-plus ~'m-plus}))
```

Named Monads

```
; create a named macro definition
(defmacro defmonad
   ([name doc-string operations]
    (let [doc-name (with-meta name {:doc doc-string})]
      `(defmonad ~doc-name ~operations)))

   ([name operations]
    `(def ~name (monad ~operations))))
```

Example: Identity Monad

```
; Identity monad
(defmonad identity-m
   "Monad describing plain computations. This monad
    does in fact nothing at all. It is useful for
    testing, for combination with monad transformers,
    and for code that is parameterized with a monad."

  [m-result identity
   m-bind   (fn m-result-id [mv f]
               (f mv))
  ])
```

Example: Writer Monad

```
(defn writer-m
  "Monad describing computations that accumulate
   data on the side, e.g. for logging. The monadic
   values have the structure [value log]. Any of the
   accumulators from clojure.contrib.accumulators can
   be used for storing the log data. Its empty value
   is passed as a parameter."
  [empty-accumulator]
  (monad
     [m-result  (fn m-result-writer [v]
                  [v empty-accumulator])
      m-bind    (fn m-bind-writer [mv f]
                  (let [[v1 a1] mv
                        [v2 a2] (f v1)]
                    [v2 (combine a1 a2)]))
     ]))
```

Monadic Expressions

```
(defmacro with-monad
    “Evaluates an expression after replacing the
     keywords defining the monad operations by the
     functions associated with these keywords in the
     monad definition given by name.”
   [name & exprs]
   `(let [~'m-bind   (:m-bind ~name)
          ~'m-result (:m-result ~name)
          ~'m-zero   (:m-zero ~name)
          ~'m-plus   (:m-plus ~name)]
      (do ~@exprs)))
```

Monadic Functions

```
(defmacro defmonadfn
  “Like defn, but for functions that use monad
   operations and are used inside a with-monad block.”
  ([name doc-string args expr]
    (let [doc-name (with-meta name {:doc doc-string})]
      `(defmonadfn ~doc-name ~args ~expr)))

  ([name args expr]
   (let [fn-name (symbol (str *ns*) (format "m+%s+m" (str name)))]
   `(do
      (defmacro ~name ~args
        (list (quote ~fn-name)
              '~'m-bind '~'m-result '~'m-zero '~'m-plus
              ~@args))
      (defn ~fn-name [~'m-bind ~'m-result ~'m-zero ~'m-plus ~@args]
        ~expr)))))
```

Example: Abstraction

```
(defn divides? [n d] (zero? (rem n d)))

(defn divisors [n]
  (let [ ds  (filter (partial divides?)
                     (range 2 (inc (int (Math/sqrt n)))))
         ds2 (map (partial / n) ds)
        ]
    (concat ds (reverse ds2))))


(defmonadfn get-divisors [n]
  (reduce m-plus m-zero (map m-result (divisors n))))
```

Example: Abstraction (cont'd)

```
user=> (with-monad maybe-m (get-divisors 23))
nil

user=> (with-monad maybe-m (get-divisors 24))
2

user=> (with-monad sequence-m (get-divisors 24))
(2 3 4 6 8 12)

user=> (with-monad sequence-m (get-divisors 25))
(5 5)

user=> (with-monad set-m (get-divisors 25))
#{5}
```

domonad

```
(fn [m rec e]
  (domonad m
           [args (m-result (rest e))
            x    (rec (first args))
            y    (rec (second args))]
           (+ x y)))
```

```
; -- Equivalent Haskell code
; \m rec e -> do let args = tail e
;                    x ← rec (args!!0)
;                    y ← rec (args!!1)
;                    return (x + y)
```

m-lift

Clojure has a lift macro that supports any number of arguments – nice!

`(m-lift 2 fn)` is equivalent to

```
(fn [arg1 arg2] (domonad [x arg1
                          y arg2]
                         (fn x y)))
```

and `(m-lift 3 fn)` is equivalent to

```
(fn [arg1 arg2 arg3] (domonad [x arg1
                               y arg2
                               z arg3]
                              (fn x y z)))
```

*In Haskell, these are separate functions `liftM`, `liftM2`, `liftM3`, etc.

Exercise: Implement mapm

The mapm function is used to map a monadic function over a list of values. It returns its list of results in the monad. It's type would be:

$(a \text{ -> } b_{\text{effect}}) \text{ -> } \textit{list-of-a} \text{ -> } (\textit{list-of-b})_{\text{effect}}$

Hint: Clojure's `reduce` function is a left fold.

maybe-t (simplified)

```
(defn maybe-t
  “Monad transformer that transforms a monad m
   into a monad in which the base values can be
   invalid (represented by nil).”
  [m]
  (monad [m-result (with-monad m m-result)
          m-bind   (with-monad m
                     (fn m-bind-maybe-t [mv f]
                       (m-bind mv
                               (fn [x]
                                 (if (identical? x nil)
                                   (m-result nil)
                                   (f x))))))
         ]))
```

state-t (simplified)

```
(defn state-t
  "Monad transformer that transforms a monad m into a
   monad of stateful computations that have the base
   monad type as their result."
  [m]
  (monad [m-result (with-monad m
                     (fn m-result-state-t [v]
                       (fn [s]
                         (m-result [v s]))))
          m-bind   (with-monad m
                     (fn m-bind-state-t [stm f]
                       (fn [s]
                         (m-bind (stm s)
                                 (fn [[v ss]]
                                   ((f v) ss))))))
         ]))
```

Lifting Operations

Lifting is tedious and error-prone. The transformer stack must be managed carefully to avoid becoming unmaintainable.

```
; we will use a (state-t (cont-t (env-t error-m)))
; stack, abbreviated scee
(def scee (state-t (cont-t (env-t error-m))))

; helper functions for lifting from different points
; in the stack

(def lift-cont (lift-state-t
                 (cont-t (env-t error-m))))

(def lift-env (comp lift-cont
                    (lift-cont-t (env-t error-m))))

(def lift-error (comp lift-env lift-env-t))
```

Why did we extend clojure.contrib.monads?

- We wanted a modular, monadic interpreter example
  - Effect requirements: errors, mutable state, continuations, logging and an environment
- This requires new monads
  - error-m
  - env-m [for completeness and testing]
- Also new monad transformers
  - cont-t
  - env-t
  - writer-t

Interpreter modularity requirements

- Our interpreter languages are built out of plug-and-play language fragments
- The interpreter monad is refined independently of the language fragments
- Changing the monad stack should affect the fragments and their assembly as little as possible
- Unfortunately, direct top-level functions as monad operations are not modular
- Their implementation must change when the structure of the monad stack changes

Implementation possibilities

- Different functions for each monad stack variation?
  - Combinatorial explosion of operation variants
  - Operation variants need to be matched to monads (difficult and error-prone)
- Better alternative: operation lifting
  - *Compute* the changes in monad operations when the transformer stack changes
  - Haskell implements lifting with typeclasses
  - Problem: this is a dynamically typed setting

Solution

- Add some indirection – monadic operations are now retrieved via the monad structure
- with-monad, domonad, etc. make these structure fields available as bound identifiers
- Monad transformers implement “uniform operation lifting” behind-the-scenes
  - This depends on some carefully-chosen auxiliary functions
- Modified library at: monad-tutorial/clojure/newmonads.clj
- This library is not polished (yet) – it just met our requirements for this tutorial example

New monad operations

- Error monad - m-fail
- State monad – m-get, m-put
- Environment monad – m-capture-env, m-local-env
- Continuation monad – m-call-cc
- Writer monad – m-write, m-listen, m-censor
- All optional, like m-zero and m-plus

- What happens if you have more than one kind of the same monad in the stack?
  (e.g. state-t (state-m))

Exercise: A Logging Fragment

- This builds on the modular interpreter infrastructure at:
  monad-tutorial/clojure/interp.clj
- Exercise instructions / framework at:
  monad-tutorial/exercises/Interpreter/log-interp.clj
- Exercise goals:
  - Writing a language fragment monadically
  - Seeing how to assemble an interpreter from pieces
  - Adding a new effect using a pre-built monad transformer