Monad Transformers and Modular Interpreters

Paper by: Sheng Liang, Paul Hudak, Mark Jones

Motivation

- Isolate PL features for better understanding and simplification
- Create language features at a high level

High-Level View of Interpreter

```
type Term = OR TermA -- arithmetic
          ( OR TermF -- functions
          ( OR TermR -- assignment
          ( OR TermL -- lazy evaluation
          ( OR TermT -- tracing
          ( OR TermC -- callcc
               TermN -- nodeterminism
            )))))

type InterpM = StateT Store  -- memory cells
             ( EnvT          -- environment
             ( ContT Answer  -- continuations
             ( StateT String -- trace output
             ( ErrorT        -- error reporting
               List          -- multiple results
               ))))

type Value = OR Int (OR Fun())
```

Monadic Interpreter?!?

- Conventional interpreter maps term, environment, and store to an answer
- **Monadic interpreters** map terms to *computations* , the environment, store, etc. are 'hidden' within the monad class structure

```
interp :: Term -> InterpM Value -- interpreter monad
```

Simple Arithmetic Language

```
type Value = OR Int () -- ()  is unit type
type Term = TermA
type InterpM = ErrorT Id
```

Examples:

```
> ((1 + 4) * 8)
40
> (3/0)
ERROR: divide by 0
```

Adding Function Calls

```
type Value = OR Int (OR Fun ())
type Term = OR TermF TermA
type InterpM = EnvT Env (ErrorT Id)
```

Examples:

```
> ((\x.(x+4))7)
11
> (x+4)
ERROR: unbound variable: x
```

Why Monads?

"Monads are nothing more than a good example of data abstractions. But they just happen to be a particularly *good* abstraction"

"Constructor Classes" in Gofer

- Useful for dealing with multiple instances of monads and monad transformers

Example:

```
class Functor f where
        map :: (a->b) -> f a -> f b

instance Functor List where
        map f []     = []
        map f (x:xs) = f x : map f xs

instance Functor Tree where
        map f (Leaf x)   = Leaf(f x)
        map f (Node l r) = Node(map f l)(map f r)
```

Union Types and Subtyping Relations

Union

```
data OR a b = L a | R b -- Inject L/R
```

Subtype

```
class SubType sub sup where
        inj :: sub -> sup                -- injection
        prj :: sup -> Maybe sub          -- projection

data Maybe a = Just a | Nothing          -- ...
```

Subtyping Relations cont.

```
instance SubType a (OR a b) where
         inj       = L
         prj(L x) = Just x
         prj _     = Nothing

instance SubType a b => SubType a (OR c b) where
         inj       = R · inj
         prj(R x) = prj a
         prj _     = Nothing
```

Interpreter Building Blocks

In order to characterize Terms we use

```
class InterpC t where
        interp :: t -> InterpM Value
```

When characterizing unions

```
instance(InterpC t1, InterpC t2) =>
                          InterpC(Or t1 t2) where
        interp(L t) = interp t
        interp(R t) = interp t
```

Before jumping into more blocks

The interpreter monad *InterpM* comes with two operations:

```
unit :: a -> InterpM a
bind :: InterpM a -> (a ->InterpM b) -> InterpM b
```

Examples:

```
> unit x
x
> m 'bind' k
runs m and pases the rest of the computation k
```

Arithmetic Building Block

Arithmetic sublanguage is given by:

```
data TermA = Num Int
           | Add Term Term
```

Monadic Interpretation

```
instance InterpC TermA where
        interp(Num x) = unitInj x
        interp(Add x y) = interp x 'bindPrj' \i ->
                          interp y 'bindPrj' \j ->
                  uniInj ((i+j)::Int)

unitInj = unit inj

m 'bindPrj' k = m 'bind' \a -> case(prj a) of
                          Just x -> kx
                          Nothing -> err "run-time type error"

err :: String -> InterpM a -- defined later
```

Function Building Block

```
data TermF = Var Name
           | LambdaN Name Term -- call by name
           | LambdaV Name Ter  -- call by value
           | App Term Term
```

Environments!

Also assuming that there's a type *Env* of environments which associate variable names with computations (closures)

```
lookupEnv :: Name -> Env -> Maybe (interpM Value)
extendEnv :: (Name, InterpM Value) -> Env -> Env
type Name = String
```

And these operations on *Env* types

```
rdEnv :: InterpM Env -- returns current environment
inEnv :: Env->InterpM a -> InterpM a
        -- performs a computation within a given environment
```

Monadic Interpretation of Functional Block

```
instance InterpC TermF where
  interp (Var v)        = rdEnv 'bind' \env ->
                          case lookupEnv v env of
                            Just val  -> val
                            Nothing   -> err ("unbound variable: " ++ v)
  interp (LambdaN s t)  = rdEnv 'bind' \env ->
                          unitInj (\arg -> inEnv (extendEnv (s, arg) env) (interp t))
  interp (LambdaV s t)  = rdEnv 'bind' \env ->
                          unitInj (\arg -> arg 'bind' \v ->
                                           inEnv (extendEnv (s, unit v) env) (interp t))
  interp (App e1 e2)    = interp e1 'bindPrj' \f ->
                          rdEnv 'bind' \env ->
                          f (inEnv env (interp e2))
```

References and Assignments

```
data TermR = Ref Term
| Deref Term
| Assign Term
```

Given a heap of memory and the following three functions, we can interpret this block

```
allocLoc :: InterpM Loc
lookupLoc :: Loc -> InterpM Value
updateLoc :: (Loc, InterpM Value) -> InterpM()
type Loc = Int
```

Interpretation

```
instance InterpC TermR where
  interp (Ref x) =
            interp x 'bind' \val →
            allocLoc 'bind' \loc →
            updateLoc (loc, unit val) 'bind' \_ →
            unitInj loc
  interp (Deref x) =
            interp x 'bindPrj' \loc →
            lookupLoc loc

  interp (Assign lhs rhs) =
            interp lhs 'bindPrj' \loc →
            interp rhs 'bind' \val →
            updateLoc (loc, unit val) 'bind' \_ →
            unit val
```

Lazy Evaluation

Block:

```
data TermL = LambdaL Name Term
```

Interpretation:

```
instance InterpC TermL where
  interp (LambdaL s t) =
    rdEnv 'bind' \env →
    unitInj (\arg →
      allocLoc 'bind' \loc →
      let thunk = arg 'bind' \v →
                  updateLoc (loc, unit v) 'bind' \_ →
                  unit v
      in
      updateLoc (loc, thunk) 'bind' \_ →
      inEnv (extendEnv (s, lookupLoc loc) env)
            (interp t))
```

Program Tracing

If we have the write function, we can define tracing!

```
write :: String -> InterpM ()
```

Block:

```
data TermT = Trace String Term
```

Interpretation:

```
instance InterpC TermT where
  interp (Trace l t) =
    write ("enter " ++ l) 'bind' \_ ->
    interp t 'bind' \v ->
    write ("leave " ++ l ++ " with:" ++ show v) 'bind' \_ ->
    unit v
```

Continuations

Block:

```
data TermC = CallCC

callcc :: ((a -> InterpM b) -> InterpM a) -> InterpM a
```

Interpretation:

```
instance InterpC TermC where
  interp CallCC = unitInj (\f ->
    f `bindPrj` \f' ->
    callcc (\k -> (f' (unitInj (\a -> a `bind` k)))))
```

Nondeterminism

Block:

```
data TermN = Amb[Term]
```

If we have a merge function, we can express the interpretation as..

```
merge :: [InterpM a] -> InterpM a

instance InterpC TermN where
        interp (Amb t) merge (map interp t)
```

Monad Transformers

Any type constructor *t* such that if *m* is a monad, so is *t m*

- They add operations to a monad
- Compose easily

```
class (Monad m, Monad(t m)) => MonadT t m where
        lift :: m a -> t m a
            -- embeds computation in monad m into monad "t m"
```

State Monad Transformer

```
type StateT s m a = s -> m(s,a) -- where m is a monad
```

Transformer

```
instance (Monad m, Monad(StateT s m)) =>
                        MonadT(StateT s) m where
        lift m = \s -> m 'bind' \x -> unit(s, x)
```

In order to support state updates

```
class Monad m => StateMonad s m where
        update :: (s -> s) -> m s

instance Monad m => StateMonad s (StateT s m) where
        update f = \s -> unit(f s, s)
```

Environment Monad Transformer

```
type EnvT r m a = r -> m a -- where m is a monad
```

Transformer

```
class Monad m => EnvMonad r (EnvT r m) where
        inEnv r m = \_ -> m r
        rdEnv = \r -> unit r

instance Monad m => EnvMonad r (EnvT r m) where
        inEnv r m = \_ -> m r
        rdEnv = \r -> unit r
```

Error Monad Transformer

```
data Error a = Ok a | Error String
type ErrorT m a = m(Error a)
```

Transformer

```
class Monad m => ErrMonad m where
        err :: String -> m a

instance Monad m => ErrMonad(ErrorT m) where
        err = unit · Error
```

Continuation Monad Transformers

```
type ContT ans m a = (a -> m ans) -> m ans
```

Transformer

```
class Monad m => ContMonad m where
        callcc :: ((a -> m b) -> m a) -> m a

instance Monad m => Monad(ContT ans m) where
        unit x = \k -> k x
        m 'bind' f = \k -> m(\a -> f a k)
```

Lists

- Cannot have a monad transformer since the other monads are not *communtative monads*
- Instead, we use lists as a *base* monad, upon which transformers can be applied

```
instance Monad List where
        unit x           = [x]
        [] `bind` k      = []
        (x:xs) `bind` k = k x ++ (xs `bind` k)

class Monad m => ListMonad m where
        merge :: [m a] -> m a

instance ListMonad List where
        merge = concat
```

Lifting

- Lifting on operation *f* in monad *m* through a monad transformer *t* results in an operation whose type signature can be derived by subbing all occurences of *m* in the type of *f* with *"t m"*

Example:

```
inEvn :: r -> m a -> m a'' -- lifted through transformer t

''r -> tm a -> t m a''    -- results in an op with this sig
```

Lifting Cont.

Given the types of operations in monad m:

$$
\begin{array}{rcll}
\tau & ::= & A & \text{(type constants)} \\
 & | & a & \text{(type variables)} \\
 & | & \tau \to \tau & \text{(function types)} \\
 & | & (\tau, \tau) & \text{(product types)} \\
 & | & m\ \tau & \text{(monad types)}
\end{array}
$$

$\lceil \rceil_t$ is the mapping of types across the monad transformer t:

$$
\begin{array}{lcl}
\lceil A \rceil_t & = & A \\
\lceil a \rceil_t & = & a \\
\lceil \tau_1 \to \tau_2 \rceil_t & = & \lceil \tau_1 \rceil_t \to \lceil \tau_2 \rceil_t \\
\lceil (\tau_1, \tau_2) \rceil_t & = & (\lceil \tau_1 \rceil_t, \lceil \tau_2 \rceil_t) \\
\lceil m\ \tau \rceil_t & = & t\ m\ \lceil \tau \rceil_t
\end{array}
$$

Lifting Types

$$
\begin{array}{lcll}
\mathcal{L}_\tau & :: & \tau \to \lceil \tau \rceil_t & \\
 & & & \\
\mathcal{L}_A & = & id & (1) \\
\mathcal{L}_a & = & id & (2) \\
\mathcal{L}_{\tau_1 \to \tau_2} & = & \backslash f \to f' \ such\ that & \\
 & & \quad f' \cdot \mathcal{L}_{\tau_1} = \mathcal{L}_{\tau_2} \cdot f & (3) \\
\mathcal{L}_{(\tau_1, \tau_2)} & = & \backslash (a, b) \to (\mathcal{L}_{\tau_1}\ a, \mathcal{L}_{\tau_2}\ b) & (4) \\
\mathcal{L}_{m\ \tau} & = & lift \cdot (map\ \mathcal{L}_\tau) & (5)
\end{array}
$$

Easy Lift Cases

```
instance(ErrMonad m, MonadT t m) => ErrMonad(t m) where
        err = lift · err


instance(StateMonad m, MonadT t m) =>  StateMonad(t m) where
        update = lift · update


instance MonadT t List => ListMonad(t List) where
        merge = join · lift
```

Conclusions

- Shown how a modular interpreter can be designed using
  - extensible union types
  - monad transformers
- Provided insights on how to design and implement extensible languages

ALSO!

I'm starting to understand monads a bit better now

Questions?

Thanks for listening!