A Modular Interpreter Built with Monad Transformers

Wolfram Kahl

April 2, 2003

Abstract

In their POPL '95 paper [Liang *et al.*, 1995], Liang, Hudak, and Jones show how to construct modular interpreters using monad transformers in the language Gofer. Nowadays, the Haskell variant Gofer is not very widespread anymore, and the original source code seems to be not easily available anymore. Since that paper also glosses over quite a few details, we here present a version of the modular interpreter that works with today's Hugs system, using its "-98" extensions.

Contents

1 Introduction

This module uses the following Hugs extensions to Haskell 98:

- multi-parameter classes, and
- overlapping instances.

Therefore, using the following command-line is necessary:

```
hugs -98 +o Interpreter.hs
```

For compilation with GHC (currently there is no `main` function) the following flags are necessary:

```
ghc -fallow-overlapping-instances  -fallow-undecidable-instances \
    -fglasgow-exts -package data -c Interpreter.lhs
```

Relative to the paper, the following are the most important changes:

- recursive type definitions have been resolved into newtype definitions,
- a few type annotations were necessary to have the monad transformers resolved correctly
- a few other types have been converted to newtype for enabling custom instances
- `Show` instances for `Term` and `Value`
- a "run" function has been added
- `ListT` (and thus nondeterminism) does not yet work — the InterpM type definition needs to be split and `newtype`d

```
module Interpreter where

import FiniteMap
```

2 Monad Basics

The type class monad is defined in the Haskell prelude as follows:

```
class Monad m where
  return :: a -> m a
  ...
```

Here, the type variable `m` is used as a one-argument type constructor, accepting a standard type and returning a standard type, so `m` has the kind `* -> *`.

Prelude monads...

STFun...

Error...

3 Terms and Interpretation

3.1 Terms — Abstract Syntax

```
type Term0 = Either TermA   -- arithmetic
           ( Either TermF   -- functions
           ( Either TermR   -- references and assignment
           ( Either TermL   -- lazy evaluation
           ( Either TermT   -- tracing
           ( Either TermC   -- callcc
           ( Either TermN   -- nondeterminism
                    ()
           ))))))

type Term1 = Either TermA   -- arithmetic
           ( Either TermF   -- functions
           ( Either TermR   -- references and assignment
           ( Either TermL   -- lazy evaluation
           ( Either TermT   -- tracing
           ( Either TermC   -- callcc
                    ()
           )))))

newtype Term = Term Term0

instance SubType a Term0 => SubType a Term where
  inj = Term . inj
  prj (Term t) = prj t
```

3.2 A Simple Subtyping Mechanism

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

```
instance SubType a (Either a b) where
  inj          = Left
  prj (Left x) = Just x
  prj _        = Nothing

instance SubType a b => SubType a (Either c b) where
  inj           = Right . inj
  prj (Right y) = prj y
  prj _         = Nothing
```

3.3 Example Terms

```
num i       = Term $ Left $ Num i
add x y     = Term $ Left $ Add x y
mult x y    = Term $ Left $ Mult x y

var x       = Term $ Right $ Left $ Var x
apply f x   = Term $ Right $ Left $ App f x
lambdaN n t = Term $ Right $ Left $ LambdaN n t
lambdaV n t = Term $ Right $ Left $ LambdaV n t

ref x       = Term $ Right $ Right $ Left $ Ref x
deref x     = Term $ Right $ Right $ Left $ Deref x
assign v t  = Term $ Right $ Right $ Left $ Assign v t

lambdaL n t = Term $ Right $ Right $ Right $ Left $ LambdaL n t

term4       = Term . Right . Right . Right . Right
trace s t   = term4 $ Left $ Trace s t
ccc         = term4 $ Right $ Left $ CallCC
ambig ts    = term4 $ Right $ Right $ Left $ Amb ts

t1 = ((num 5 `add` num 3) `mult` num 2) `add` num 7

t2 = apply (lambdaL "x" t1') (num 5 `add` num 8)
t1' = ((num 5 `add` var "x") `mult` num 2) `add` num 7

t2'' = apply (lambdaL "x" t1'') (num 5 `add` num 8)
t1'' = (trace "add5" (num 5 `add` var "x") `mult` num 2) `add` num 7
t2'  = apply (lambdaL "x" t1') (ambig [num 5, num 7] `add` num 8)

Interpreter> interpret1 ( ccc `apply` (lambdaN "f" (var "f" `apply` (var "f" `apply` num 5))))

@@> callcc (\_f.f (f 5))


@@> = 5
```

3.4 Values

```
type Value0 = Either Integer (Either Fun ())

newtype Value = Value Value0


newtype Fun = Fun (InterpM Value -> InterpM Value)
unFun (Fun x) = x


instance SubType a Value0 => SubType a Value where
  inj = Value . inj
  prj (Value x) = prj x


instance Show Value where
  showsPrec _ (Value (Left i)) = shows i
  showsPrec _ (Value (Right (Left f))) = shows f
  showsPrec _ _ = ("()" ++)

instance Show Fun where
  showsPrec _ f s = "<function>" ++ s
```

3.5 The Class of Interpretable Terms

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

instance (InterpC t1, InterpC t2) => InterpC (Either t1 t2) where
  interp (Left  t) = interp t
  interp (Right t) = interp t

instance InterpC Term where
  interp (Term t) = interp t

instance InterpC () where
  interp () = error "illegal term"
```

3.6 Auxiliary Monad Functions: Overview

Here just a summary of the auxiliary functions that are used for interpretation — they will be defined by the various monadic building blocks later. All of them are available on `InterpM` — for some of them it did not make sense to preserve a modular type.

Errors:

```
| err       :: ErrMonad m   => String -> m a
```

Nondeterminism:

```
| merge     :: ListMonad m  => [m a] -> m a
```

Environment:

```
| rdEnv     :: EnvMonad e m => m e
| inEnv     :: EnvMonad e m => e -> m a -> m a
```

Store locations:

```
| allocLoc  ::                 InterpM Loc
| lookupLoc ::                 Loc -> InterpM Value
| updateLoc ::                 (Loc,InterpM Value) -> InterpM ()
```

Tracing output:

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

Continuations:

```
| callcc    :: ContMonad m  => ((a -> m b) -> m a) -> m a
```

3.7 Arithmetic Expressions

```
data TermA = Num Integer   | Add Term Term | Mult Term Term
           | Sub Term Term | Div Term Term | Mod  Term Term

arithBinopInterm op x y = interp x `bindPrj` \ i ->
                          interp y `bindPrj` \ j ->
                          returnInj ((i `op` j) :: Integer)

instance InterpC TermA where
  interp (Num x) = returnInj x
  interp (Add x y) = arithBinopInterm (+) x y
  interp (Mult x y) = arithBinopInterm (*) x y
  interp (Sub x y) = arithBinopInterm (-) x y
  interp (Div x y) = arithBinopInterm div x y
  interp (Mod x y) = arithBinopInterm mod x y


returnInj :: (Monad m, SubType a b) => a -> m b
returnInj = return . inj

bindPrj :: (SubType a b, ErrMonad m) => m b -> (a -> m d) -> m d
bindPrj m k = do a <- m
                 case prj a of
                   Just x -> k x
                   Nothing -> err "run-time type error"
```

3.8 Functions

```
data TermF = Var Name
           | LambdaN Name Term
           | LambdaV Name Term
```

```
          | App Term Term

type Name = String

lookupEnv :: Name -> Env -> Maybe (InterpM Value)
extendEnv :: (Name, InterpM Value) -> Env -> Env

newtype Env = Env (FiniteMap Name (InterpM Value))
lookupEnv n (Env e) = lookupFM e n
extendEnv (n,v) (Env e) = Env (addToFM e n v)

instance InterpC TermF where
  interp (Var v) = do env <- rdEnv
                      case lookupEnv v env of
                        Just val -> val
                        Nothing -> err ("unbound variable: " ++ v)
  interp (LambdaN s t) =
        do env <- rdEnv
           returnInj $ Fun (\ arg ->
                         inEnv (extendEnv (s,arg) env) (interp t))

  interp (LambdaV s t) =
        do env <- rdEnv
           returnInj $ Fun (\ arg ->
               do v <- arg
                  inEnv (extendEnv (s, return v) env) (interp t))

  interp (App e1 e2) = interp e1 `bindPrj` \ f ->
                       do env <- rdEnv
                          unFun f (inEnv (env :: Env) (interp e2))
```

3.9 References and Assignment

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

allocLoc  ::                          InterpM Loc
lookupLoc :: Loc                   -> InterpM Value
updateLoc :: (Loc, InterpM Value) -> InterpM ()

type Loc = Integer

instance InterpC TermR where
  interp (Ref x) = do val <- interp x
                      loc <- allocLoc
                      updateLoc (loc, return val)
                      returnInj loc

  interp (Deref x) = interp x `bindPrj` lookupLoc
```

```
  interp (Assign lhs rhs) = interp lhs `bindPrj` \ loc ->
                            interp rhs >>= \ val ->
                            updateLoc (loc, return val) >>
                            return val

-- allocLoc :: StateMonad Store m => m Loc
allocLoc = liftSTFun allocStore

lookupLoc i = join $ liftSTFun (lookupStore i)

updateLoc p = liftSTFun (updateStore p)

data Store = Store {next :: Integer,
                    store :: FiniteMap Integer (InterpM Value)}

allocStore (Store n fm) = (Store (n+1) fm, n)
lookupStore i s = (s, lookupWithDefaultFM (store s) e i)
  where e = error ("illegal store access at " ++ show i)
updateStore (i, v) (Store n fm) = (Store n (addToFM fm i v), ())
```

3.10 Lazy Evaluation

```
data TermL = LambdaL Name Term

instance InterpC TermL where
  interp (LambdaL s t) =
    do env <- rdEnv
       returnInj $ Fun ( \arg ->
         do loc <- allocLoc
            let thunk = do v <- arg
                           updateLoc (loc, return v)
                           return v
            updateLoc (loc, thunk)
            inEnv (extendEnv (s, lookupLoc loc) env)
                  (interp t))
```

3.11 Tracing

```
data TermT = Trace String Term

instance InterpC TermT where
  interp (Trace l t) = do write ("enter " ++ l)
                          v <- interp t
                          write ("leave " ++ l ++ " with: " ++ show v)
                          return v

write :: String -> InterpM ()
write msg = do update (\ sofar -> sofar ++ [msg])
               return ()
```

3.12 Continuations

```
data TermC = CallCC

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

instance InterpC TermC where
  interp CallCC = returnInj $ Fun
      ( \ f -> f `bindPrj` \ g ->
                callcc ( \ k -> unFun g ( returnInj $ Fun (>>= k))))
```

3.13 Nondeterminism

```
data TermN = Amb [Term]

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

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

4 Monad Transformers

4.1 Additional Monad Functions

```
mmap :: Monad m => (a -> b) -> m a -> m b
mmap f m = do a <- m; return (f a)

join :: Monad m => m (m a) -> m a
join z = z >>= id
```

4.2 Monad Transformers

```
class (Monad m, Monad (t m)) => MonadT t m where
  lift :: m a -> t m a
```

Folgende *Gesetze* müssen gelten:

```
-- lift . return = return
-- lift (m >>= k) = lift m >>= (lift . k)
```

4.3 Error Monad Transformer

```
data Error a = Ok a | Error String

newtype ErrorT m a = ErrorT (m (Error a))

unErrorT (ErrorT x) = x

instance Monad m => Monad (ErrorT m) where
```

```
  return = ErrorT . return . Ok
  fail msg = ErrorT $ return (Error msg)
  ErrorT m >>= k = ErrorT $ do a <- m
                               case a of
                                 Ok x -> unErrorT (k x)
                                 Error msg -> return (Error msg)


instance Monad m => MonadT ErrorT m where
  lift = ErrorT . mmap Ok

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

instance Monad m => ErrMonad (ErrorT m) where
  err = ErrorT . return . Error
```

4.4 State Monad Transformer

```
newtype StateT s m a = StateT (s -> m (s,a))

unStateT (StateT x) = x

instance Monad m => Monad (StateT s m) where
  return x = StateT (\ s -> return (s,x))

  m >>= k = StateT (\ s0 -> do (s1,a) <- unStateT m s0
                               unStateT (k a) s1)


instance Monad m => MonadT (StateT s) m where
  lift m = StateT (\ s -> do x <- m; return (s,x))


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


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



liftSTFun :: StateMonad s m => (s -> (s,a)) -> m a
liftSTFun f = do s <- update id
                 let (s',a) = f s
                 update (const s')
                 return a

getState :: StateMonad s m => m s
```

```
getState = update id

putState :: StateMonad s m => s -> m ()
putState s = do update (const s); return ()
```

4.5 The Entire Interpreter Monad

```
type InterpM = StateT Store       -- store locations
             ( EnvT Env           -- environment
             ( ContT Answer       -- continuations
             ( StateT [String]    -- tracing
             ( ErrorT             -- error handling
               []                 -- nondeterminism
             ))))

type Answer = Value
```

4.6 Environment Monad Transformer

```
newtype EnvT r m a = EnvT (r -> m a)

unEnvT (EnvT x) = x


instance Monad m => Monad (EnvT r m) where
  return a = EnvT (\ r -> return a)

  EnvT m >>= k = EnvT (\ r -> do a <- m r
                                 unEnvT (k a) r)


instance Monad m => MonadT (EnvT r) m where
  lift m = EnvT (\ r -> m)


class Monad m => EnvMonad env m where
  inEnv :: env -> m a -> m a
  rdEnv :: m env


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

4.7 Continuation Monad Transformer

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

unContT (ContT x) = x
```

```
instance Monad m => Monad (ContT ans m) where
  return x = ContT (\ k -> k x)

  ContT m >>= f  =
        ContT (\ k -> m ( \ a -> unContT (f a) k))


instance Monad m => MonadT (ContT ans) m where
  lift m = ContT (m >>=)


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


instance Monad m => ContMonad (ContT ans m) where
  -- callcc :: ((a -> ContT ans m b) -> ContT ans m a) -> ContT ans m a
  -- f :: (a -> ContT ans m b) -> ContT ans m a
  -- k :: (a -> m ans) -> m ans
  callcc f = ContT (\ k -> unContT (f (\ a -> ContT (\ _ -> k a))) k)
```

4.8 List Monads

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

instance ListMonad [] where
  merge = concat
```

5 Propagation of Special Operations over Monad Transformers

5.1 Implementation of `EnvMonad` via `StateMonad`

```
{-
instance (Monad m, StateMonad r m) => EnvMonad r m where
  inEnv r m = do o <- update (const r)
                 v <- m
                 update (const o)
                 return v
  rdEnv = update id
-}
```

5.2 Simple Propagation

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

instance (StateMonad s m, MonadT t m) => StateMonad s (t m) where
```

```
  update = lift . update

instance (ListMonad m, MonadT t m) => ListMonad (t m) where
  merge = error "ListMonad undefined" -- join . lift

instance MonadT t [] => ListMonad (t []) where
  merge = join . lift
```

5.3 Propagation of EnvMonad

```
instance (MonadT (EnvT r') m, EnvMonad r m) =>
                              EnvMonad r (EnvT r' m) where
  rdEnv = lift rdEnv

  inEnv r (EnvT m) = EnvT (\ r' -> inEnv r (m r'))


instance (MonadT (StateT s) m, EnvMonad r m) =>
                               EnvMonad r (StateT s m) where
  rdEnv = lift rdEnv

  inEnv r (StateT m) = StateT (\ s -> inEnv r (m s))


instance (MonadT ErrorT m, EnvMonad r m) =>
                           EnvMonad r (ErrorT m) where
  rdEnv = lift rdEnv

  inEnv r m = inEnv r m


instance (MonadT (ContT ans) m, EnvMonad r m) =>
                                EnvMonad r (ContT ans m) where
  rdEnv = lift rdEnv

  -- inEnv :: env -> ContT ans m a -> ContT ans m a
  -- c :: (a -> m ans) -> m ans
  -- k :: a -> m ans
  inEnv (r :: env) (ContT c) =
     ContT (\ k -> do o <- rdEnv
                      inEnv r (c (inEnv (o :: env) . k)))


{- Alternative:

instance (MonadT (ContT ans) m, EnvMonad r m) =>
                                EnvMonad r (ContT ans m) where
  rdEnv     = lift rdEnv

  inEnv r (ContT c) = ContT (\ k -> inEnv r (c k))
```

```
-}
```

5.4 Propagation of ContMonad

```
instance (MonadT (EnvT r) m, ContMonad m) => ContMonad (EnvT r m) where
  -- callcc :: ((a -> EnvT r m b) -> EnvT r m a) -> EnvT r m a
  -- f :: (a -> EnvT r m b) -> EnvT r m a

  callcc f = EnvT (\ r ->
     callcc ( \ k -> unEnvT (f ( \ a -> EnvT (\ _ -> k a))) r))


instance (MonadT (StateT s) m, ContMonad m) => ContMonad (StateT s m) where
  -- callcc :: ((a -> StateT s m b) -> StateT s m a) -> StateT s m a
  -- f :: (a -> StateT s m b) -> StateT s m a

  callcc f = StateT (\ s0 -> callcc
          (\ k -> unStateT (f (\ a -> StateT (\ s1 -> k (s1,a)))) s0))


instance (MonadT ErrorT m, ContMonad m) => ContMonad (ErrorT m) where
  -- callcc :: ((a -> ErrorT m b) -> ErrorT m a) -> ErrorT m a

  callcc f = ErrorT
    (callcc ( \ k -> unErrorT (f (\ a -> ErrorT (k (Ok a))))))
```

6 Running the Interpreter

```
run :: InterpM Value -> [Error ([[Char]],Value)]

run (StateT f) = let
  EnvT g = f (Store 0 emptyFM)
  ContT h = g (Env emptyFM)
  StateT i = h (\ (store,v) -> return v)
  ErrorT j = i []
 in j


interpret :: Term -> [Error ([[Char]],Value)]
interpret = run . interp

myShowFirstResult (Ok (trace,v) : _ ) =
  foldr h ("\n@@> = " ++ show v) trace
    where h tr r = tr ++ '\n' : r

myShowFirstResult (Error msg : _) = msg


interpret1 t = putStrLn ("\n@@> " ++ shows t ('\n' : '\n' :
```

```
                        myShowFirstResult (interpret t)))
```

Examples:

```
dupN = lambdaN "x" (var "x" `add` var "x")
dupV = lambdaV "x" (var "x" `add` var "x")
dupL = lambdaL "x" (var "x" `add` var "x")
tdupN = trace "dupN" dupN
tdupV = trace "dupV" dupV
tdupL = trace "dupL" dupL

tr2 = trace "2" (num 2)

ff2 lambda = (lambda "f" (var "f" `apply` (var "f" `apply` tr2)))

fN = ff2 lambdaN `apply` tdupN
fV = ff2 lambdaV `apply` tdupV
fL = ff2 lambdaL `apply` tdupL

Interpreter> dup `apply` num 2
(\_x.x+x) 2

Interpreter> interpret (dup `apply` num 2)
[([],4)]

Interpreter> interpret1 (tdupN `apply` tr2)

@@> (trace (\_x.x+x)) (trace 2)

enter dupN
leave dupN with: <function>
enter 2
leave 2 with: 2
enter 2
leave 2 with: 2

@@> = 4

Elapsed time (ms): 20 (user), 0 (system)
Interpreter> interpret1 (tdupV `apply` tr2)

@@> (trace (\!x.x+x)) (trace 2)

enter dupV
leave dupV with: <function>
enter 2
leave 2 with: 2

@@> = 4
```

```
Elapsed time (ms): 30 (user), 0 (system)
Interpreter> interpret1 (tdupL `apply` tr2)

@@> (trace (\x.x+x)) (trace 2)

enter dupL
leave dupL with: <function>
enter 2
leave 2 with: 2

@@> = 4

Interpreter> interpret1 (ff2 lambdaN `apply` tdupN)

@@> (\_f.f (f (trace 2))) (trace (\_x.x+x))

enter dupN
leave dupN with: <function>
enter dupN
leave dupN with: <function>
enter 2
leave 2 with: 2
enter 2
leave 2 with: 2
enter dupN
leave dupN with: <function>
enter 2
leave 2 with: 2
enter 2
leave 2 with: 2

@@> = 8


Interpreter> interpret1 (ff2 lambdaV `apply` tdupV)

@@> (\!f.f (f (trace 2))) (trace (\!x.x+x))

enter dupV
leave dupV with: <function>
enter 2
leave 2 with: 2

@@> = 8

Interpreter> interpret1 (ff2 lambdaL `apply` tdupL)

@@> (\f.f (f (trace 2))) (trace (\x.x+x))

enter dupL
leave dupL with: <function>
```

```
enter 2
leave 2 with: 2

@@> = 8

Interpreter> interpret1 (lambdaV "y" (num 42) `apply` (ff2 lambdaN `apply` tdupN))

@@> (\!y.42) ((\_f.f (f (trace 2))) (trace (\_x.x+x)))

enter dupN
leave dupN with: <function>
enter dupN
leave dupN with: <function>
enter 2
leave 2 with: 2
enter 2
leave 2 with: 2
enter dupN
leave dupN with: <function>
enter 2
leave 2 with: 2
enter 2
leave 2 with: 2

@@> = 42


Interpreter> interpret1 (lambdaN "y" (num 42) `apply` (ff2 lambdaN `apply` tdupN))

@@> (\_y.42) ((\_f.f (f (trace 2))) (trace (\_x.x+x)))


@@> = 42

instance Show TermA where
  showsPrec p (Num x) = shows x
  showsPrec p (Add x y) = (if p > 5 then paren else id)
                          (showsPrec 5 x . ('+' : ) . showsPrec 5 y)
  showsPrec p (Mult x y) = (if p > 6 then paren else id)
                           (showsPrec 6 x . ('*' : ) . showsPrec 6 y)

instance Show TermF where
  showsPrec _ (Var v) = (v ++)
  showsPrec 0 (LambdaN n t) = ('\\':). ('_':) . (n ++) . ('.':) . shows t
  showsPrec p l@(LambdaN n t) = paren (showsPrec 0 l)
  showsPrec 0 (LambdaV n t) = ('\\':) . ('!':) . (n ++) . ('.':) . shows t
  showsPrec p l@(LambdaV n t) = paren (showsPrec 0 l)
  showsPrec p (App x y) = (if p > 1 then paren else id)
                          (showsPrec 1 x . (' ' : ) . showsPrec 9 y)

paren f = ('(' : ) . f . (')' : )
```

```
instance Show TermR where
  showsPrec p (Ref t) = ("ref " ++) . showsPrec 9 t
  showsPrec p (Deref t) = ('!':) . showsPrec 9 t
  showsPrec p (Assign v t) = (if p > 0 then paren else id)
                             (showsPrec 0 v . (" := " ++) . showsPrec 1 t)

instance Show TermL where
  showsPrec 0 (LambdaL n t) = ('\\':) . (n ++) . ('.':) . shows t
  showsPrec p l@(LambdaL n t) = paren (showsPrec 0 l)

instance Show TermT where
  showsPrec p (Trace s t) = (if p > 0 then paren else id)
                            (("trace " ++) . showsPrec 9 t)

instance Show TermC where
  showsPrec p CallCC = ("callcc" ++)

instance Show TermN where
  showsPrec p (Amb ts) = showList ts

showsPrecTerm n (Left t) = showsPrec n t
showsPrecTerm n (Right t1) = case t1 of
  Left t -> showsPrec n t
  Right t2 -> case t2 of
   Left t -> showsPrec n t
   Right t3 -> case t3 of
    Left t -> showsPrec n t
    Right t4 -> case t4 of
     Left t -> showsPrec n t
     Right t5 -> case t5 of
      Left t -> showsPrec n t
      Right t6 -> showsPrec n t6

instance Show Term where
  showsPrec n (Term t) = showsPrecTerm n t

instance Show a => Show (Error a) where
  showsPrec _ (Ok x) = shows x
  showsPrec _ (Error msg) = ("error: " ++) . (msg ++)
```

6.1 The Identity Monad

```
newtype Id a = Id a
instance Monad Id where
  return = Id
  Id x >>= f  =  f x

instance SubType a (Id a) where
  inj = Id
  prj (Id x) = Just x
```

7 Exercise

Extend the interpreter by adding Boolean values and expressions, and `if _ then _ else _` expressions.

References

[Liang *et al.*, 1995] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *22nd POPL*. acm press, 1995.