universität innsbruck

WS 2024/2025

Advanced Functional Programming

Week 6 – Evaluation of Monadic Code, Example: Tseitin, Error Monads

René Thiemann

Department of Computer Science

Last Week

- monads in general
  - aim: convenient chaining of computations
  - `return` and `(>>=)` can be user-defined: programmable semicolon
  - monad laws must be satisfied
  - do-notation
  - example monads: `Maybe`, `State`, `IO`
- state monad
  - encapsulate state
  - purely functional: `State s a` is roughly `s -> (a,s)`
  - or using IO: `newIORef`, `readIORef`, `writeIORef`
- example: randomized quicksort
  - advantage `IO`: a bit faster than `State` and potentially perfect RNG
  - advantage `State`: no side effects, final result is pure function
- in general there is a disadvantage of using `IO`
  - function of type `... -> IO a` can have arbitrary side effects
  - function of type `... -> State s a` can at most alter state of type `s`

Evaluation of Monadic Code

Evaluation of Monadic Code

- consider the following Haskell code

```haskell
g b = putStrLn (show b) >> return b

f mb1 mb2 = do
  b1 <- mb1
  b2 <- mb2
  return $ b1 || b2
```

- result of `f (g True) (g False)` (IO monad)
  - both `putStrLn` will be executed, since both monadic operations will be executed, even if `b1 || b2` will not look at `b2`
- result of `evalState (f (return True) (error "foo")) ()` (State monad)
  - lazy evaluation will figure out that the final state is not required, result is `True` without any error message
- result of `f (return True) (error "foo") :: Maybe Bool` (Maybe monad)
  - bind of `Maybe` is strict, so computation is aborted with error `"foo"`
- overall: evaluation of monadic code highly depends on chosen monad

Evaluation of Monadic Code, Another Example

- consider the following Haskell code

```
h m1 m2 m3 = do
  x <- m1
  y <- m2
  z <- m3
  return (x, y, z)

test1 = let xs = Just [1..100 :: Int] in h xs xs xs

test2 = let xs =      [1..100 :: Int] in h xs xs xs
```

- result of `test1`
  - Just ([1..100], [1..100], [1..100]) — `Maybe` monad
- result of `test2`
  - a list of all possible triples with numbers between 1 and 100 — `List` monad
- overall: evaluation of monadic code highly depends on chosen monad

Example: Memoization of Embedding Relation, Handling Memoization

- we setup generic code for computing the embedding relation in a monadic way

```
embMain :: (Eq f, Eq v, Monad m) =>
  (Term f v -> Term f v -> m (Maybe Bool)) -- lookup
  -> (Term f v -> Term f v -> Bool -> m ()) -- store
  -> Term f v -> Term f v -> m Bool
embMain look store = main where
  main s t = do
    maybeResult <- look s t
    case maybeResult of
      Just b -> return b
      Nothing -> do
        result <- main2 s t
        store s t result
        return result
```

- `main` just does the handling of memory-lookups and memory-stores
- `main2` will perform the actual computation

Example: Memoization of Embedding Relation, Main Algorithm

- remaining code of `embMain` looks like the definition of the embedding relation

```
  main2 (Var x) t = return $ t == Var x
  main2 (Fun f ss) t@(Fun g ts)
   | f == g = do
      bigConj <- allM ( \ (si,ti) -> main si ti) (zip ss ts)
      bigDisj <- anyM ( \ si -> main si t) ss
      return $ bigConj || bigDisj
  main2 (Fun f ss) t = anyM ( \ si -> main si t) ss

allM, anyM :: Monad m => (a -> m Bool) -> [a] -> m Bool
```

- `allM`, `anyM` are monadic variants of `all, any :: (a -> Bool) -> [a] -> Bool`
- here: illustrate two variants how to achieve this lifting via `mapM` and `foldM`

```
allM f xs = and <$> mapM f xs
anyM f xs = foldM (\ b x -> (b ||) <$> f x) False xs
```


Example: Memoization of Embedding Relation, Wrapper using IO and State

- finally, we can derive two implementations via `IO` or via `State`

```
embState :: (Ord f, Ord v) => Term f v -> Term f v -> (Bool, Int)
embState s t = let
    look s t = return . M.lookup (s,t) =<< get
    store s t b = put . M.insert (s,t) b =<< get
    (res, m) = runState (embMain look store s t) M.empty
  in (res, M.size m)

embIO :: (Ord f, Ord v) => Term f v -> Term f v -> IO (Bool, Int)
embIO s t = do
 ref <- newIORef M.empty
 let look s t = return . M.lookup (s,t) =<< readIORef ref
 let store s t b = writeIORef ref . M.insert (s,t) b =<< readIORef ref
 res <- embMain look store s t
 m <- readIORef ref
 return (res, M.size m)
```

Execution of Memoized Embedding Implementations

- consider execution time of `embIO s t` or `embState s t` for some test terms `s` and `t`
  - `embIO s t` 1.77 seconds
  - `embState s t` 1.45 seconds
- now let us only access the Boolean result (ignore size of the map)
  - `fst <$> embIO s t` 1.80 seconds
  - `fst $ embState s t` 0.16 seconds
- reason: `State` monad can profit from lazy evaluation, `IO` cannot
  - as soon as the Boolean result is determined, all pending `put`-commands can be ignored in the `State` monad
  - using `IO`, each `writeIORef` operation must be performed
- solution to discrepancy: design some lazy monadic operations (exercises)

Example Application: Tseitin Transformation

More Complex Setups

- often, several values need to be stored and updated globally
  - state for generating next fresh name, state for some dictionaries, . . .
- common solution: use one datatype as state with many entries and use record syntax
- moreover, one might require features of several monads
- common solution: make monad features abstract by using type classes
- setup of Haskell's state monad in `Control.Monad.State` as type class

```
class Monad m => MonadState s m  where
  get :: m s
  put :: s -> m ()

gets :: MonadState s m => (s -> a) -> m a  -- get with selector function
modify :: MonadState s m => (s -> s) -> m ()

{- type "State" is just one instance of class "MonadState" -}
```


Example: Tseitin Transformation

- algorithm to convert propositional formula into conjunctive normal form (CNF)
  - input: arbitrary Boolean formula (conjunction, disjunction, negation, variables)
  - first, label each non-variable subformula by some fresh propositional variable
  - second, encode that fresh propositional variables have correct values by using small CNFs
  - finally, demand that fresh propositional variable at root evaluates to true
  - result: obtain equi-satisfiable CNF of linear size
- requirements on state monad
  - encode (fresh) variables as integers (convention in standard Dimacs format for CNFs)
  - state has to store a single number for next fresh variable
  - moreover, original variables need to be mapped to integers, too;
    so, state needs a map from original variables to integer variables

Tseitin Transformation in Haskell – Datatypes

```haskell
data Formula a =
   Conj [Formula a]
 | Disj [Formula a]
 | Neg (Formula a)
 | Var a
  deriving Show

type CnfVar = Integer           -- negative sign = negated variable
type VarMap a = M.Map a CnfVar

type Clause = [CnfVar]

data TseitinState a = TseitinState {
     lastUsedCnfVar :: CnfVar,
     varMap :: M.Map a CnfVar
  }
```


Tseitin Transformation in Haskell – Auxiliary Functions

```haskell
nextCnfVar :: MonadState (TseitinState a) m => m CnfVar
nextCnfVar = do
  x <- gets lastUsedCnfVar             -- access state via record selector
  let fresh = x + 1
  modify (\ s -> s { lastUsedCnfVar = fresh }) -- modify via record update
  return fresh

lookupVar :: (Ord a, MonadState (TseitinState a) m) => a -> m CnfVar
lookupVar x = do
  vmap <- gets varMap
  case M.lookup x vmap of
    Just i -> return i
    Nothing -> do
       i <- nextCnfVar
       modify (\ s -> s { varMap = M.insert x i vmap })
       return i
```


Two Observations

- adding more elements to `TseitinState` will neither require changes to `lookupVar` nor to `nextCnfVar`
  - reason: both functions use record syntax, and this syntax does not change when adding more elements to `TseitinState`
- the class constraints are not of standard shape
  - `nextCnfVar :: MonadState (TseitinState a) m => m CnfVar` expresses that we need a monad state with a specific type as state (`TseitinState a`)
  - such a type-class constraint is not allowed w.r.t. the Haskell 2010 standard
  - consequence: activate GHC extension `{-# FlexibleContexts #-}`

Tseitin Transformation in Haskell – Main Algorithm

```haskell
addClause :: MonadWriter [Clause] m => Clause -> m ()
addClause c = tell [c]

tseitinMain ::
  (Ord a, MonadState (TseitinState a) m, MonadWriter [Clause] m) =>
  Formula a -> m CnfVar

tseitinMain (Var x) = lookupVar x
tseitinMain (Disj fs) = do
   fis <- mapM tseitinMain fs
   j <- nextCnfVar
   addClause $ - j :  fis                 -- CNF encoding of j -> (\/ fis)
   mapM_ (\ fi -> addClause [j, - fi]) fis -- CNF encoding of (\/ fis) -> j
   return j
-- Conj and Neg: similar to Disj
```

Remarks and Final Version

- `MonadWriter` is another type of monad, that allows users to produce output via `tell :: MonadWriter w m => w -> m ()`; collect output after running monad
- resulting algorithm `tseitinMain` is very close to text book; all the tedious implementation details are delegated to the monad
- wrapper around `tseitinMain` just needs to find a monad that satisfies all of the monadic class constraints
- one possibility: `RWS`, the reader-writer-state monad

```
tseitin :: Ord a => Formula a -> ([Clause], Integer, M.Map a CnfVar)
tseitin f =
  let initS = TseitinState {lastUsedCnfVar = 0, varMap = M.empty}
  in case runRWS (tseitinMain f) () initS of
    (fIndex, finalState, clauses) ->
      let allClauses = [fIndex] : clauses
          nrVariables = lastUsedCnfVar finalState
          mapping = varMap finalState
        in (allClauses, nrVariables, mapping)
```


Final Remarks

- RWS combines reader-, writer- and state-monad
- state monad has been discussed thoroughly
- reader monad (`Control.Monad.Reader`)
  - monad stores common read-only environment
  - `ask :: MonadReader r m => m r`
  - environment is fixed when running monad
- writer monad (`Control.Monad.Writer`)
  - monad stores produced output
  - `tell :: MonadWriter w m => w -> m ()`
  - produced output becomes accessible after running monad
- for further information, see Haskell documentation
  - https://hackage.haskell.org/package/mtl/docs/Control-Monad-Reader.html
  - https://hackage.haskell.org/package/mtl/docs/Control-Monad-Writer.html
  - https://hackage.haskell.org/package/mtl/docs/Control-Monad-State.html
  - https://hackage.haskell.org/package/mtl/docs/Control-Monad-RWS.html

Error Monads

Error Monads

- main purpose: encapsulate computations that may fail
- example applications: parsing, type checking, accessing dictionaries, ...
- example monads
  - `Maybe`
    - instance: `return = Just; Nothing >>= _ = Nothing; Just x >>= f = f x`
    - representing a failure: `Nothing`
  - `Either e` (`data Either e a = Left e | Right a`)
    - instance: `return = Right; Left e >>= _ = Left e; Right x >>= f = f x`
    - representing a failure with explicit error: `Left e`
  - `IO a`
    - instance: built-in
    - representing a failure with error message: `error msg`
- convention: all of these monads should treat their error-handling in the same monad, e.g., do not use `error` in `Maybe` or `Either e` to indicate a failure

Example Application: Find Carrier Billing Address

- scenario: given several maps, do a compositional lookup
  - use name to find phone number
  - use phone number to find mobile carrier
  - use mobile carrier to find billing address
- setup in Haskell importing `Data.Map` as `M`

```
type PersonName = String
type PhoneNumber = String
type BillingAddress = String

data MobileCarrier = Honest_Bobs_Phone_Network | ... deriving (Eq, Ord)

findCarrierBillingAddress :: PersonName
  -> M.Map PersonName PhoneNumber
  -> M.Map PhoneNumber MobileCarrier
  -> M.Map MobileCarrier BillingAddress
  -> Maybe BillingAddress
```


Find Carrier Billing Address: Version 1

```
fCBAversion1 person phoneMap carrierMap addressMap =
    case M.lookup person phoneMap of
      Nothing -> Nothing
      Just number ->
        case M.lookup number carrierMap of
          Nothing -> Nothing
          Just carrier -> M.lookup carrier addressMap
```

- explicit case analysis, no use of monad operations
- this is the style of programming that we would like to avoid

Versions 2 and 3 use Maybe-monad and do-Notation

```
fCBAversion2 person phoneMap carrierMap addressMap = do
  number <- M.lookup person phoneMap
  carrier <- M.lookup number carrierMap
  address <- M.lookup carrier addressMap
  return address

fCBAversion3 person phoneMap carrierMap addressMap = do
  number <- M.lookup person phoneMap
  carrier <- M.lookup number carrierMap
  M.lookup carrier addressMap
```

- much cleaner code
- version 2 is more canonically: every lookup is done in the same way
- optimization in version 3: last lookup can directly return final result

Versions 4 and 5: Point-free Versions

```
fCBAversion4 person phoneMap carrierMap addressMap =
    lookup phoneMap person >>= lookup carrierMap >>= lookup addressMap
  where lookup :: Ord k => M.Map k v -> k -> Maybe v
        lookup = flip M.lookup
```

- point-free: intermediate results are not stored, but directly passed to next function
- requires shuffling of arguments of `M.lookup` so that search-key is last argument
- similar to nested function applications, which often start on rhs
  `idea:  lookup addressMap $ lookup carrierMap $ lookup phoneMap person`
- to allow composition in this order, use flipped version of (>>=)

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b

fCBAversion5 person phoneMap carrierMap addressMap =
    lookup addressMap =<< lookup carrierMap =<< lookup phoneMap person
```

Do-Notation and Error-Monads

- idea of translations of do-blocks

```
do x <- m              =        m >>= \ x -> do block
   block

do m                   =        m >> do block
   block

do let x = e           =        let x = e in do block
   block
```

- what should be result of `secondProblem (return "a")` for

```
secondProblem m = do (_ : x : _) <- m
                     return x
```

  - runtime exception complaining about incomplete pattern?
  - `Nothing`, if the chosen monad is `Maybe`?
  - `Left ???`, if the chosen monad is `Either e`?

Do-Notation and Error-Monads Continued

- design choice: unmatched patterns in do-block must be resolved by failure type of monad
- consider program again

```
secondProblem m = do (_ : x : _) <- m
                     return x
```

  - `secondProblem (return "a" :: IO String)` leads to runtime exception
  - `secondProblem (return "a" :: Maybe String)` results in `Nothing`
  - `secondProblem (return "a" :: Either String String)` leads to compile error
- note type of program: `secondProblem :: MonadFail m => m [a] -> m a`
- `MonadFail` extends `Monad` and contains a failure function
  `fail :: String -> m a`
- `IO` and `Maybe` are instances of `MonadFail`
- `Either e` is not an instance of `MonadFail`: how to convert `String` to `e`?
- details
  - https://hackage.haskell.org/package/base/docs/Control-Monad-Fail.html
  - https://gitlab.haskell.org/haskell/prime/-/wikis/libraries/proposals/monad-fail

Do-Notation and Error-Monads Finalized

- reconsider transformation of do-blocks

```
-- if p always matches
do p <- m   =   m >>= (\ p -> do block)
   block

-- if p might fail
do p <- m   =   m >>= (\ x -> case x of { p -> do block; _ -> fail msg})
   block
```

- to prevent enforcement of `MonadFail`, one can indicate that a pattern will always match
  - `~pat` is the irrefutable pattern that always matches
  - only if variable bindings in `pat` are used, then the matching substitution is computed and runtime errors might occur

```
secondProblem2 :: Monad m => m [a] -> m a   -- no restriction on monad m,
secondProblem2 m = do ~(_ : x : _) <- m     -- secondProblem2 (return "a")
                      return x              --   always results in error

f (x : ~(y : _)) = x || y       -- f [True] = True, f [False] = error
```


Exercise Task 1 – Improve the Implementation of the Embedding Relation

(A) Improve the monadic implementation, so that there is no significant time difference between all four presented variants, i.e., using `IO` or `State` with or without computation of size.
To this end, think about how to make Boolean operations (disjunction, conjunction, all, any) lazy in their monadic versions. In particular, the size of the final map should become significantly smaller by your optimizations.

(B) Use the labeling of terms that has been developed in the exercise of the previous week. Change the type of the dictionary so that labels are used as keys instead of the full terms. What changes?

Exercise Task 2 – Improve the Implementation of the Tseitin Transformation

The function `tseitin` runs in quadratic time, since the writer part of the currently used monad `RWS ... [Clause] ...` uses lists in the output part, and each `tell cl` will append a clause `cl` to the end of the output list. So standard lists are not a good choice as the `w`-parameter for a writer in this application.

Note that `w` can be an arbitrary `Monoid`, cf. `https://hackage.haskell.org/package/mtl/docs/Control-Monad-Writer-Lazy.html`.
Figure out a better monoid, so that `tseitin` can be reformulated (without changing `tseitinMain`) so that it runs in linear time. Either define your own instance of a monoid, or use an existing one from a suitable library.
If possible, your changes should not influence the generated CNF. To be more precise, running `testInvocation` should result in the same string, no matter whether you use `tseitin` from `Demo06_Tseitin_Monad_RWS` or from `Exercise06_Tseitin`.

Literature

- Functional Programming with Overloading and Higher-Order Polymorphism, Mark P Jones, Advanced School of Functional Programming, 1995.
- Real World Haskell, Chapters 14 and 15