Advanced Programming 2022

Monads, continued

Andrzej Filinski

andrzej@di.ku.dk

Department of Computer Science
University of Copenhagen

September 15, 2022

Where are we?

- First week of AP:
  - Generally useful Haskell features: type classes, laziness/reasoning, functional I/O, list comprehensions.
- Last time:
  - Monads as encapsulating particular *notions of effects*.
    - Errors/exceptions (`Maybe`, `Either` e)
    - State variants (`State` s, `Reader` r, `Writer` w)
    - Lists/nondeterminism (`[]`)
  - Type constructor + `return` + `(>>=)` + monad-specific associated operations
- Today:
  - Interaction trees (`IO`, `SimpleIO`), choice trees
  - Combining monads

Monadic I/O

- Defined some time time ago:

```haskell
data SimpleIO a = Done a  -- like "[]", when a = ()
                | PutChar Char (SimpleIO a) -- "Char : [Char]"
                | GetChar (Char -> SimpleIO a)
```

- Sequence (actually tree) of instructions for top-level printer.
- Can organize as [so-called *free*] monad, by specifying how to "append" request sequences (trees):

```haskell
instance Monad SimpleIO where
  return a            = Done a
  (Done a)      >>= f = f a  -- "[] ++ f = f"
  (PutChar c m) >>= f = PutChar c (m >>= f)
                             -- "(c : m) ++ f = c : (m ++ f)"
  (GetChar h)   >>= f = GetChar (\c -> h c >>= f)
```

- With associated operations:

```haskell
simplePutChar :: Char -> SimpleIO ()
simplePutChar c = PutChar c (return ())  -- "c : []"

simpleGetChar :: SimpleIO Char
simpleGetChar = GetChar (\c -> return c)
```

I/O, continued

- Can define further I/O operations by normal monadic sequencing:

```
simplePutStr :: String -> SimpleIO ()
simplePutStr s = mapM_ simplePutChar s
  -- using mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
  -- mapM_ f [a1,...,an] = do f a1; ...; f an; return ()
```

- In particular, can check (by unfolding the defs) that `simplePutStr "AP"` $\simeq$ `PutChar 'A' (PutChar 'P' (Done ()))`

- Can also turn `SimpleIO` computations into "real" `IO` actions:

```
perform :: SimpleIO a -> IO a
perform (Done a)      = return a
perform (PutChar c m) = do putChar c; perform m
perform (GetChar h)   = do c <- getChar; perform (h c)
```

Choice trees

- `SimpleIO` tree branches based on *keyboard (stdin) input*.
  - Monads let us construct this tree while writing program in natural, quasi-imperative style.
- Can also branch based on nondeterminstic *guesses*.
  - $\approx$ computation in List monad
    - `do ...; guess <- pick [p1, p2, p3]; ...`
    - Branching nodes contain *lists* of subtrees (`[Tree]`), not *functions* (`Char -> Tree`).
  - Determines (potentially infinite) tree of choices, constructed on demand.
  - Analogous to *game tree* in `TicTacToe` from Exercise set 1.
  - *Explore* tree with any standard search/optimization algorithm:
    - Depth-first (simple backtracking)
    - Breadth-first (avoids infinite fruitless paths)
    - Best-first (explore promising paths first)
    - Branch-and-bound (prune away suboptimal parts)
  - Separates *search stratgy* from (often complex) *domain modeling*.
  - `May` say more about this in case studies at end of course.

Combining monads

- Have seen a range of *basic* monadic effects.
  - Maybe 2–3 more are commonly used, of similar complexity.
- But how do we deal with programs that use *multiple* effects?
  - In general, create *custom-tailored* monad for any particular combination of effects. Order may matter!
- **Example:** exceptions and (*persistent*) state

```
newtype ExnState s e a =
  ExSt {runExSt :: s -> (Either e a, s)}
instance Monad (ExnState s e) where
  return a = ExSt (\s -> (Right a, s))
  m >>= f = ExSt (\s -> case runExSt m s of
                        (Left e, s') -> (Left e, s')
                        (Right a, s') -> runExSt (f a) s')

putState :: s -> ExnState s e ()
putState s' = ExSt (\s -> (Right (), s'))

throwExn :: e -> ExnState s e a
throwExn e = ExSt (\s -> (Left e, s))
```

Combining monads, continued

- Also possible to combine exceptions and state with a *transactional* semantics.
  - State modifications *discarded* when error signaled.
- Subtle modifications of monad type and operations:

```
newtype StateExn s e a = StEx {runStEx:: s -> Either e (a,s)}
instance Monad (StateExn s e) where
  return a = StEx (\s -> Right (a, s))
  m >>= f = StEx (\s -> case runStEx m s of
                          Left e -> Left e
                          Right (a, s') -> runStEx (f a) s')
  -- could also express using components of (Either e) monad

putState :: s -> StateExn s e ()
putState s' = StEx (\s -> Right ((), s'))

throwExn :: e -> StateExn s e a
throwExn e = ExSt (\s -> Left e)
```

Monad transformers

- GHC comes with *Monad Transformer Library (MTL)*
  - Systematic way of building complex monads out of elementary building blocks.
  - A *monad transfomer* extends a monad with new features.
  - Example: state monad-transfomer:

    ```
    newtype Monad m =>
             StateT s m a = St {runSt :: s -> m (a, s)}
    ```
  - Then `type State s a` $\simeq$ `StateT s Identity a`
    - where `newtype Identity a = Id a` is the *identity* monad
  - And `type StateExn s e a` $\simeq$ `StateT s (Either e) a`
    - Also need to *lift* the monad operations from `m` to `StateT s m`.
  - See details and docs on Hoogle (search for, e.g., `StateT`)
- For *AP*, quite manageable to build combined monads by hand.
  - We'll *not* expect you to use monad transformers.
  - But you are welcome to use them where relevant (and respecting other constraints of the task).