1. **The Hindley-Milner Type System and Inference Algorithm**

1.1. **Type inference.**
   - Trade-off between expressiveness and efficiency/(decidability) of checking.
   - Type system vs. type inference algorithm.
   - Can get limited form of System F’s parametric polymorphism without having to write the types by hand. (Would you want to write these annotations for more complicated values?)

\[
T = \Lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. x
\]
\[
F = \Lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. y
\]
\[
3 = \Lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \rightarrow \alpha. f (f \ x)
\]

- Sometimes have to reason about type inference limitations and algorithm to understand type errors.
- Annotating type signatures can sometimes help a program type-check.

1.2. **Hindley-Milner.**
   - Algorithm has two steps, generating the equations and solving the equations (via unification).
   - The delta from *simple types* is type schemes:

\[
\sigma ::= \tau
\]
\[
\mid \forall t. \sigma
\]

- We *generalize* a type to introduce polymorphism by capturing the new type variables.

\[
\text{Gen(TE, } \tau \text{)} = \forall t_1 \ldots t_n. \tau, \{t_1 \ldots t_n\} = \text{FV}(\tau) - \text{FV(TE)}
\]

  – Generalize occurs only for let statements.

- A type scheme is *instantiated* by substituting types for bound variables.
  – Instantiation occurs during unification.
  – A *generic instance* maps to new type variables.

2. **Monads**

2.1. **In class this week.**
   - Using monads for features outside the scope of the language (IO).
   - Using monads as a useful abstraction for things expressible within the language (gensym).
   - To come: the theory behind monads. (In category theory, a monad is a functor from a category to itself.)

2.2. **Monads in Haskell.**
   - IO.

```haskell
getLine :: IO String
getLine = do c <- getChar
            if c == '\n'
              then return ""
              else do l <- getLine
                      return (c : l)
```

- Do-block syntax does not have to be whitespace-sensitive.
getLine :: IO String
getLine = do { c <- getChar; if c == '\n' then return "" else do {...}}

- Do is just syntactic sugar.
  - Then (>>) operator.
    do e1 ; e2 = e1 >>= e2
  - Bind (>>=) operator.
    do p <- e1; e2 = e1 >>= \p -> e2

2.3. Haskell Monad class. We can define our own monads in Haskell.

infixl 1 >>>, >>=
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a

    m >>= k = m >>= \_ -> k

2.4. The Maybe monad. A simple but useful monad.

data Maybe a = Nothing | Just a

instance Monad Maybe where
    return = Just
    fail = Nothing
    Nothing >>= f = Nothing
    (Just x) >>= f = f x

instance MonadPlus Maybe where
    mzero = Nothing
    Nothing `mplus` x = x
    x `mplus` _ = x