MAXWELL SWADLING

EXTENDED USES OF TEMPLATE META-PROGRAMMING

YOW! Lambda Jam 2014

EXTENDED META-PROGRAMMING

- Construct proofs
- Inference
- Create extensible data structures

- Tools:
  - Template Haskell
  - Constraint solver

TEMPLATE HASKELL

- Boilerplate elimination
- Code generation
- Quasi Quoter

```haskell
data Banana = Banana
  { shape :: Field "banana-shape" Text
  , size  :: Field "banana size" (Maybe Int)
  , name  :: Field "banana's name" Text
  } deriving Show

deriveToJSONFields ''Banana


b = Banana (Field "foo") (Field (Just 2))
(Field "bar")

-- >> encode b
-- "{\"banana's name\":\"bar\",\"banana size
\":2,\"banana-shape\":\"foo\"}"
```

LABELLED AESON

```haskell
newtype Field (n :: Symbol) v = Field { unField :: v }
  deriving Show

deriveToJSONFields ty = do
  t <- reify ty
  case t of
    TyConI (DataD _ _ ts [cs] _) -> do
     let (n, cs') = case cs of
                      NormalC n xs -> (n, [t | (_, t)    <- xs])
                      RecC n xs ->    (n, [t | (_, _, t) <- xs])
```

LABELLED AESON

```
instance ToJSON Banana where
  toJSON (Banana a_1 a_2 a_3)
    = object
        [(.=) "banana-shape" a_1,
         (.=) "banana size" a_2,
         (.=) "banana's name" a_2]
```

n: Name of constructor

cs': Types of fields

```
    fs <- sequence [(,) (fieldName x) `fmap` newName "a" | x <- cs']
    sequence [instanceD (return []) (appT (conT ''ToJSON) (conT ty)) [
        funD 'toJSON [clause [conP n (map (varP . snd) fs)] (normalB (
            appE (varE 'object) (listE [
              appE (appE (varE '(.=)) (litE (StringL fieldN)))
                   (varE fieldVar)
            | (fieldN, fieldVar) <- fs ])
          )) []]
      ]]
  _ -> error "single constr only for now"
where
  fieldName :: Type -> String
  fieldName (AppT (AppT (ConT _Name) (LitT (StrTyLit s))) _) = s
```

QUASI QUOTER

```haskell
-- [digitQ|4|] :: Digit
-- 4
--
-- named [digitQ|4|]  = "four"
-- named [digitQ|$x|] = "not four, " ++ show x ++ " instead"
--
-- mod10D x = let y = mod x 10 in [digitQ|$y|]

digitQ :: QuasiQuoter
digitQ = QuasiQuoter {
    quoteExp = dexp
  , quotePat = dpat
  , quoteType = error "not quotable"
  , quoteDec = error "not quotable"
  }

dexp :: [Char] -> ExpQ
dexp ('$':vn) = varE (mkName vn)
dexp (d:[])   = maybe (error "not a digit”)
                    (dataToExpQ (const Nothing)) (d ^? digitC)
dexp _        = error "not a digit"

dpat :: [Char] -> PatQ
dpat ('$':vn) = varP (mkName vn)
dpat (d:[])   = maybe (error "not a digit”)
                    (dataToPatQ (const Nothing)) (d ^? digitC)
dpat _        = error "not a digit"
```

CONSTRAINT SOLVER

- Type class (constraint)
- Type function

CONSTRAINT SOLVER

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

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

undefined = undefined

isF1 :: Functor f => f a
isF1 = fmap undefined undefined

isF2 :: Applicative f => f a
isF2 = fmap undefined undefined

-- isF3 :: Functor f => f a
-- isF3 = pure undefined

isF4 :: Applicative f => f a
isF4 = pure undefined
```

CONSTRAINT SOLVER

```haskell
-- kind bool
data Bool = True | False

type family Not (a :: Bool) :: Bool

type instance Not True = False
type instance Not False = True

b1 :: Not True ~ False => a
b1 = undefined

-- b2 :: Not False ~ False => a
-- b2 = undefined
```


CONSTRUCTING PROOFS

CONSTRUCTING PROOFS

- Prove things the compiler can't
- We need more axioms

CONSTRUCTING PROOFS

- Traverse the domain
- Write down axioms in type / class instances
- Type checker solves type function

EXTENDING TYPE LITS

- In 7.6, nothing worked

```
f :: ((1 + 1) ~ 2) => ()
Couldn't match type `1 + 1' with `2'
```

- In 7.8, some stuff works

```
f :: ((1 + 1) ~ 2) => ()
f :: (0 ~ (1 - 1)) => ()
```

- For everything else, proof by construction / exhaustion

ADDITION

```
type family Add (m :: Nat) (n :: Nat) :: Nat

numberSystem :: Integer -> Q [Dec]
numberSystem theBiggestNumber = return $ map (\i ->
      TySynInstD ''Add (TySynEqn [ LitT (NumTyLit i)
                                 , LitT (NumTyLit 1)
                                 ] (LitT (NumTyLit (i + 1))))
      ) [0..theBiggestNumber]
```

```
-- type instance Add 5 1 = 6
```

```
type Two = Add 1 1
```

DIVISION

```
type family Div (m :: Nat) (n :: Nat) :: Nat

numberSystem :: Integer -> Q [Dec]
numberSystem theBiggestNumber = return $ map (\i -> map (\j ->
      TySynInstD ''Div (TySynEqn [ LitT (NumTyLit (i * j))
                                 , LitT (NumTyLit i)
                                 ] (LitT (NumTyLit (j))))
      ) [0..theBiggestNumber]) [1..theBiggestNumber]
```

```
-- type instance Div 4 2 = 2
```

```
type Two = Div 4 2
```

A BIT MORE COMPLICATED

But Maxwell,
I have Peano numbers

- Numbers have inductive definitions
- A Tic Tac Toe game is not so easy

TIC TAC TOE

```
type family TICTACTOE (x1 :: CELL) (x2 :: CELL) (x3 :: CELL)
                      (y1 :: CELL) (y2 :: CELL) (y3 :: CELL)
                      (z1 :: CELL) (z2 :: CELL) (z3 :: CELL) :: GAME

data GAME = START | PROGRESS | WINNERA | WINNERB | DRAW
data CELL = NOBODY | PLAYERA | PLAYERB
```

```haskell
data THG = N | A | B | D
  deriving (Show, Eq, Ord)

newtype Gam = Gam [THG]
  deriving (Show, Eq, Ord)

 move A = conT 'PLAYERA
 move B = conT 'PLAYERB
 move N = conT 'NOBODY
 winth A = conT 'WINNERA
 winth B = conT 'WINNERB
 winth N = conT 'PROGRESS
 winth D = conT 'DRAW

tictactoe :: Q [Dec]
tictactoe = mapM gmOf $ concat
          $ map (mkGame (Gam [N, N, N, N, N, N, N, N, N]) A) [0..8]
  where
```

```
ot A = B
ot B = A
set i t gm = let
  (h, r) = splitAt i gm
  in (h ++ (t : tail r))


mkGame :: Gam -> THG -> Int -> [Gam]
mkGame (Gam gm) t i = if gm !! i /= N
  then []
  else let ng = Gam (set i t gm)
           moreg :: [Gam]
           moreg = if winner gm == N
                   then concat $ map (mkGame ng (ot t)) [0..8]
                   else []
       in nub . sort $ ((ng :: Gam) : (moreg :: [Gam]))
```

```haskell
winner gm = let
  c1 = (col 0 gm)
  c2 = (col 1 gm)
  c3 = (col 2 gm)
  r1 = (row 0 gm)
  r2 = (row 1 gm)
  r3 = (row 2 gm)
  d1 = (diL gm)
  d2 = (diR gm)
  res = catMaybes [c1, c2, c3, r1, r2, r3, d1, d2]
  in if null res
    then if any (== N) gm
      then N
      else D
    else head res
col n gm =
  if gm !! (0 + n) == A && gm !! (3 + n) == A && gm !! (6 + n) == A
  then Just A
  else if gm !! (0 + n) == B && gm !! (3 + n) == B && gm !! (6 + n) == B
    then Just B
    else Nothing
row n gm =
  if gm !! (0 + (n * 3)) == A && gm !! (1 + (n * 3)) == A && gm !! (2 + (n * 3)) == A
  then Just A
  else if gm !! (0 + (n * 3)) == B && gm !! (1 + (n * 3)) == B && gm !! (2 + (n * 3)) == B
    then Just B
    else Nothing
diL gm = if gm !! 0 == A && gm !! 4 == A && gm !! 8 == A
  then Just A
  else if gm !! 0 == B && gm !! 4 == B && gm !! 8 == B
    then Just B
    else Nothing
diR gm = if gm !! 2 == A && gm !! 4 == A && gm !! 6 == A
  then Just A
  else if gm !! 2 == B && gm !! 4 == B && gm !! 6 == B
    then Just B
    else Nothing
```

INFERENCE

INFERENCE

- If there is only one correct value, we can infer it
- Write down facts with Template Haskell
- Infer values with the Constraint Solver

TIC TAC TOE SOLVE

```
data SOLVE (a :: GAME) where
  GameStarting :: SOLVE START
  GameProgress :: SOLVE PROGRESS
  Draw         :: SOLVE DRAW
  WinnerA      :: SOLVE WINNERA
  WinnerB      :: SOLVE WINNERB
```

TIC TAC TOE SOLVE

```
class Game (a :: GAME) where
  (?) :: SOLVE a

instance Game START where
  (?) = GameStarting
instance Game PROGRESS where
  (?) = GameProgress
instance Game DRAW where
  (?) = Draw
instance Game WINNERA where
  (?) = WinnerA
instance Game WINNERB where
  (?) = WinnerB

type instance TICTACTOE NOBODY NOBODY NOBODY
                        NOBODY NOBODY NOBODY
                        NOBODY NOBODY NOBODY = START
```

TIC TAC TOE QQ

```
tq :: QuasiQuoter
tq = QuasiQuoter {
      quoteExp = error "not quotable"
    , quotePat = error "not quotable"
    , quoteType = dt
    , quoteDec = error "not quotable"
    }
  where
    dt :: String -> TypeQ
    dt s = appT (conT ''SOLVE)
         $ foldl (\x y -> appT x (conT y))
                 (conT ''TICTACTOE)
                 ((>>=) s gam)
    gam :: Char -> [Name]
    gam 'x' = ['PLAYERA]
    gam 'o' = ['PLAYERB]
    gam '?' = ['NOBODY]
    gam _ = []
```

TIC TAC TOE RESULT

```
game :: ([tq| x o x
              o o x
              □ □ x])
game = (?)
```

```
*Main> :t game
game
  :: SOLVE
       (TICTACTOE
          ‘PLAYERA ‘PLAYERB 'PLAYERA
          ‘PLAYERB ‘PLAYERB 'PLAYERA
          ‘NOBODY   ‘NOBODY 'PLAYERA)
*Main> game
WinnerA
```

DATA.TYPE.EQUALITY

```
import Data.Type.Equality

t :: ([tq| x o x
           o o x
           ? ? x |]) :~: SOLVE WINNERA
t = Refl


t :: ([tq| x o x
           o o x
           ? ? x |]) :~: SOLVE DRAW
t = Refl
```

```
Main.hs:8:5:
    Couldn't match type ‘WINNERA’ with ‘DRAW’
```

LENS

```haskell
newtype Breed = Breed { unBreed :: String }
  deriving Show

data Colour = White | Red | Sesame
  deriving Show

newtype Age = Age { unAge :: Int }
  deriving (Show, Num)

data Inu = Inu { _breed :: Breed
               , _colour :: Colour, _age :: Age }
  deriving Show
```

INU

```
kabosu :: Inu
kabosu = Inu (Breed "Shiba Inu") Red 6

kabosu_breed :: Breed
kabosu_breed = kabosu ^. breed

name :: Inu -> String
name x = "Kawaii " ++ unBreed (x ^. breed)
```

INFLENS

```
class IsInferable a b f where
  (???) :: Functor f => (b -> f b) -> a -> f a

data Foo = Foo { _bar :: String, _baz :: Int }

instance Functor f => IsInferable Foo String f where
  (???) = bar

instance Functor f => IsInferable Foo Int f where
  (???) = baz
```

INFLENS

- Create lenses with Template Haskell
- Provide instances for a type class
- Constraint Solver infers values

INU

```haskell
kabosu :: Inu
kabosu = Inu (Breed "Shiba Inu") Red 6

kabosu_breed :: Breed
-- kabosu_breed = kabosu ^. breed
kabosu_breed = kabosu ^. (???)

name :: Inu -> String
-- name x = "Kawaii " ++ unBreed (x ^. breed)
name x = "Kawaii " ++ unBreed (x ^. (???))
```

%~? ^.?

```
infixr 4 %~?
(%~?) :: IsInferable a b Identity
      => (b -> b) -> a -> a
(%~?) = (%~) (???)

infixr 4 ^.?
(^.?) :: IsInferable a b (Const b) => a -> b
(^.?) = flip (^.) (???)
```

INU

```haskell
kabosu_breed :: Breed
-- kabosu_breed = kabosu ^. breed
-- kabosu_breed = kabosu ^. (???)
kabosu_breed = (^.?) kabosu

name :: Inu -> String
-- name x = "Kawaii " ++ unBreed (x ^. breed)
-- name x = "Kawaii " ++ unBreed (x ^. (???))
name x = "Kawaii " ++ unBreed ((^.?) x)
```

INU BIRTHDAY

```
birthday :: Age -> Age
birthday (Age x) = Age (x + 1)

inu_birthday :: Inu -> Inu
-- inu_birthday = age %~ birthday
-- inu_birthday = (???) %~ birthday
inu_birthday = (%~?) birthday
```

INKO

```
data Inko = Inko { _inkoAge :: Age }
  deriving Show
makeInferableLenses ''Inko

inkoChan = Inko 4

older :: IsInferable a Age Identity => a -> a
older x = birthday %~? x
```

DATA STRUCTURES

DATA STRUCTURES

- Create extensible / flexible data structures
- Use the Constraint Solver to perform induction

MAP

- Key value map
- Safe by construction
- No Template Haskell required

MAP TYPE

```
newtype Map (k :: [Nat]) v = Map [v]
  deriving Show

empty :: Map '[] a
empty = Map []

add :: Proxy k -> v -> Map ks v -> Map (k ': ks) v
add _ v (Map xs) = Map (v:xs)
```

MAP !!

```
class KnownNat k => Ke (k :: Nat) (ks :: [Nat]) v where
  (!!) :: Proxy k -> Map ks v -> v
```

```
instance KnownNat k => Ke k (k ': ks) v where
  _ !! (Map (x:_)) = x
```

```
instance Ke k ks v => Ke k (h ': ks) v where
  k' !! (Map (_:xs)) = k' !! (Map xs :: Map ks v)
```

MAP

```
g :: Map [3, 10, 1] String
g = add (undefined :: Proxy 3)  "baz"
  $ add (undefined :: Proxy 10) "bar"
  $ add (undefined :: Proxy 1)  "foo"
  $ empty

v1 = (undefined :: Proxy 10) !! g
v2 = (undefined :: Proxy 3) !! g
v3 = (undefined :: Proxy 1) !! g
```

SYMBOL MAP

- Strings for keys
- Optional keys

SYMBOL MAP TYPE

```
data SMap (k :: [Symbol]) v = SMap [v] (M.Map String v)
  deriving Show
```

```
emptys :: SMap '[] a
emptys = SMap [] M.empty

adds :: Proxy k -> v -> SMap ks v -> SMap (k ': ks) v
adds _ v (SMap xs m) = SMap (v:xs) m
```

```
addo :: String -> v -> SMap ks v -> SMap ks v
addo k v (SMap vs m) = SMap vs $ M.insert k v m
```

SYMBOL MAP ! / !?

```
class KnownSymbol k => Ma (k :: Symbol) (ks :: [Symbol]) v where
  (!) :: Proxy k -> SMap ks v -> v
  (!?) :: Proxy k -> SMap ks v -> Maybe v

instance KnownSymbol k => Ma k (k ': ks) v where
  _ !  (SMap (x:_) _) = x
  _ !? (SMap (x:_) _) = Just x

instance Ma k ks v => Ma k (h ': ks) v where
  k' !  (SMap (_:xs) m) = k' ! (SMap xs m :: SMap ks v)
  k' !? (SMap (_:xs) m) = k' !? (SMap xs m :: SMap ks v)

instance KnownSymbol k => Ma k '[] v where
  _  !  (SMap _ _) = undefined
  k' !? (SMap _ m) = M.lookup (symbolVal k') m
```

SYMBOL MAP

```
type HTTPHeaders = SMap ["connection", "accept", "host"] String

httpIn :: HTTPHeaders
httpIn = adds (undefined :: Proxy "connection") "keep-alive"
       $ adds (undefined :: Proxy "accept")     "text/plain"
       $ adds (undefined :: Proxy "host")       "maxs.io"
       $ addo "content-length"                  "9001"
       $ emptys


m1 = (undefined :: Proxy "host") ! httpIn
m2 = (undefined :: Proxy "content-length") !? httpIn
```

SIZED VECTOR

- A vector of length n
- Add some Template Haskell

SIZED VECTOR TYPE

```
newtype MVec (l :: Nat) t = MVec { unLen :: IOVector t }

numberSystem 10

#define NAT(x) (fromIntegral (natVal (undefined :: Proxy x)))
```

```
type family Div (m :: Nat) (n :: Nat) :: Nat

numberSystem :: Integer -> Q [Dec]
numberSystem theBiggestNumber = return $ concat divs
  where
    divs = map (\i -> map (\j ->
      TySynInstD ''Div (TySynEqn [ LitT (NumTyLit (i * j))
                                 , LitT (NumTyLit i)
                                 ] (LitT (NumTyLit j)))
      ) [0..theBiggestNumber]) [1..theBiggestNumber]
```

SIZED VECTOR TAKE

```
take :: forall l m t. (KnownNat l, KnownNat m, Storable t)
     => (m <= l)
     => MVec l t -> MVec m t
take (MVec v) = MVec $ M.unsafeTake NAT(m) v
```

```
v1 :: MVec 10 Double <- replicate 1.5

take v1 :: MVec 5 Double

fromList [1.5,1.5,1.5,1.5,1.5]
```

SIZED VECTOR DROP

```
drop :: forall l m t. (KnownNat l, KnownNat m,
                       KnownNat (l - m), Storable t)
     => MVec l t -> MVec m t
drop (MVec v) = MVec $ M.unsafeDrop NAT((l - m)) v
```

```
v1 :: MVec 10 Double <- replicate 1.5
```

```
> drop v1 :: MVec 11 Double
```

```
<interactive>:9:1:
    No instance for (KnownNat (10 - 11)) arising from a use of 'drop`
    In the expression: drop v1 :: MVec 11 Double
    In an equation for it: it = drop v1 :: MVec 11 Double
```

SIZED VECTOR TAKEEACH

```
takeEach :: forall l s t. (KnownNat l, KnownNat s, Storable t)
         => Proxy s -> MVec l t -> MVec (Div l s) t
takeEach _ (MVec v) = MVec $ unsafeInlineST $ do
  x <- N.unsafeFreeze v
  let x' = N.ifilter isModZero x
  N.unsafeThaw x'
  where
    isModZero i _ = mod i NAT(s) == 0
```

```
v1 :: MVec 10 Double <- replicate 1.5
```

```
> takeEach (undefined :: Proxy 2) v1 :: MVec 5 Double
fromList [1.5,1.5,1.5,1.5,1.5]
```


WHAT ELSE?

WHEN TO USE THIS?

- Difficult inductive definition
- Need Typeable
- Convenience (Inferable)
- Extensible / flexible data structure

OTHER LANGUAGES

- Scala Shapeless
  - Miles Sabin
  - Shapeless Lens Inference
  - Map

WHAT'S NEXT?

- Limitations
- GHC as a Library

"Type a quote here."

–JOHNNY APPLESEED

https://github.com/maxpow4h/ylj-2014