Little Languages

Dave Laing

YOW! Lambda Jam 2016

Haskell is great for writing DSLs.

There are some techniques and libraries out there that can really help.

Theory helps more.

Get comfortable with the theory and reap the awesome.

Bonus points: Approach the modularity of the theory.

B

Terms (maths)

$$
\begin{aligned}
t = {} & \mathsf{true} \\
\mid {} & \mathsf{false} \\
\mid {} & \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t
\end{aligned}
$$

Terms (Haskell)

```
data Term =
    TmTrue
  | TmFalse
  | TmIf Term Term Term
```

Extending B (part 1)

An extension

```
tmNot :: Term -> Term
tmNot x = TmIf x TmFalse TmTrue

tmAnd x y :: Term -> Term -> Term
tmAnd x y = TmIf x y False

tmOr x y :: Term -> Term -> Term
tmOr x y = TmIf x True y
```

An extension

```
data Term =
    ...
  | TmNot Term
  | TmAnd Term Term
  | TmOr Term Term
```

An evaluator (?)

```
eval :: Term
     -> Term
eval TmTrue = TmTrue
eval TmFalse = TmFalse
eval (TmIf t1 t2 t3) =
  case eval t1 of
    TmTrue -> t2
    TmFalse -> t3
```

Is this correct?

What does correctness mean here?

Values

Values are the terms that don't need evaluation.

Values (maths)

$$
\begin{aligned}
v = {} & \mathsf{true} \\
\mid {} & \mathsf{false}
\end{aligned}
$$

Values (maths)

$$\frac{}{\text{true val}}\ \text{V-True}$$

$$\frac{}{\text{false val}}\ \text{V-False}$$

Values (Haskell)

```haskell
isValue :: Term
        -> Bool
isValue TmTrue =
  True
isValue TmFalse =
  True
isValue _ =
  False
```

Values (Haskell)

```
value :: Term
        -> Maybe Term
value TmTrue =
  Just TmTrue
value TmFalse =
  Just TmFalse
value _ =
  Nothing
```

$$\frac{}{\text{true val}}\ \text{V-True}$$

```haskell
valueTmTrue :: Term
            -> Maybe Term
valueTmTrue TmTrue =
  Just TmTrue
valueTmTrue _ =
  Nothing
```

$$\frac{}{\textsf{false val}}\ \textsf{V-False}$$

```
valueTmFalse :: Term
             -> Maybe Term
valueTmFalse TmFalse =
  Just TmFalse
valueTmFalse _ =
  Nothing
```

```haskell
value :: Term
      -> Maybe Term
value tm =
  valueTmTrue tm <|>
  valueTmFalse tm
```

```haskell
import Data.Foldable (asum)

asum :: (Foldable t, Alternative f)
     => t f a -> f a
```

```haskell
valueRules :: [Term -> Maybe Term]
valueRules =
  [ valueTmTrue
  , valueTmFalse
  ]

value :: Term
      -> Maybe Term
value tm =
  asum .
  fmap ($ tm) $
  valueRules
```

Small-step semantics of B

Small-step semantics (maths)

$$\frac{}{\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2}\ \text{E-IfTrue}$$

$$\frac{}{\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3}\ \text{E-IfFalse}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}\ \text{E-If}$$

If no step is defined for a term, it is a *normal form.*

All values are normal forms.

A normal form that is not a value is *stuck.*

We use `Maybe` to handle pattern match failures in the rules.

We use `Maybe` to handle the partiality of the combined step function.

$$\frac{}{\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2} \text{ E-IfTrue}$$

```
eIfTrue :: Term
        -> Maybe Term
eIfTrue (TmIf TmTrue tm2 _) =
  Just tm2
eIfTrue _ =
  Nothing
```

$$\frac{}{\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3}\ \text{E-IfFalse}$$

```
eIfFalse :: Term
         -> Maybe Term
eIfFalse (TmIf TmFalse _ tm3) =
  Just tm3
eIfFalse _ =
  Nothing
```

$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \text{ E-If}$$

```
eIf :: (Term -> Maybe Term)
    -> Term
    -> Maybe Term
eIf step (TmIf tm1 tm2 tm3) = do
  tm1' <- step tm1
  return $ TmIf tm1' tm2 tm3
eIf =
  Nothing
```

```haskell
smallStepRules :: [Term -> Maybe Term]
smallStepRules =
  [ eIfTrue
  , eIfFalse
  , eIf smallStep
  ]

smallStep :: Term
          -> Maybe Term
smallStep tm =
  asum .
  fmap ($ tm) $
  smallStepRules
```

An evaluator (!)

```
smallStepEval :: Term
              -> Term
smallStepEval tm =
  case smallStep tm of
    Nothing -> tm
    Just tm’ -> smallStepEval tm’
```

Properties of the small-step semantics

There are some properties that we want to hold for our small-step function.

Property 1

All values are normal forms.

Property 2

No terms are stuck.

Property 3

Small-step is determinate.

Property 4

Small-step is normalizing.

Testing the small-step semantics

Need some generators and friends

```
genAnyTerm :: Gen Term
shrAnyTerm :: Term -> [Term]
```

We can wrap them in newtypes

```haskell
newtype AnyTerm =
  AnyTerm {
    getAnyTerm :: Term
  } deriving (Eq, Ord, Show)

instance Arbitrary AnyTerm where
  arbitrary =
    AnyTerm <$> genAnyTerm
  shrink =
    fmap AnyTerm .
    shrAnyTerm .
    getAnyTerm
```

We can use them in properties directly

```
forAllShrink genAnyTerm shrAnyTerm $ \tm -> ...
```

We also need some helpers

```
isValue      =
  isJust . value

canStep      =
  isJust . smallStep

isNormalForm =
  isNothing . smallStep
```

We also need some helpers

```
termSize TmFalse =
  1
termSize TmTrue  =
  1
termSize (TmIf tm1 tm2 tm3) =
  1 + termSize tm1 + termSize tm2 + termSize tm3
```

Property 1

All values are normal forms.

```
prop_valueIsNormal :: AnyTerm -> Property
prop_valueIsNormal (AnyTerm tm) =
  isValue tm ==> isNormalForm tm
```

Property 2

No terms are stuck.

```
prop_normalIsValue :: AnyTerm -> Property
prop_normalIsValue (AnyTerm tm) =
  isNormalForm tm ==> isValue tm
```

Property 3

Small-step is determinate.

```haskell
prop_determinate :: AnyTerm -> Property
prop_determinate (AnyTerm tm) =
  canStep tm ==>
    let
      distinctResults =
        length .
        group .
        mapMaybe ($ tm) $
        smallStepRules
    in
      distinctResults === 1
```

Property 1-3 (and then some)

```
prop_unique :: AnyTerm -> Property
prop_unique (AnyTerm tm) =
  let
    matches =
      length .
      mapMaybe ($ tm) $
      valueRules ++ smallStepRules
  in
    matches === 1
```

Property 4

Small-step is normalizing.

```
prop_normalizing :: AnyTerm -> Property
prop_normalizing (AnyTerm tm) =
  case smallStep tm of
    Nothing -> True
    Just tm’ -> termSize tm’ < termSize tm
```

Bonus Property

```
prop_correct_fast :: AnyTerm -> Property
prop_correct_fast (AnyTerm tm) =
  smallStepEval tm === fastCowboyEval tm
```

A type system for B

```
data Type =
  TyBool
```

```haskell
inferTerm :: Term
          -> Type
inferTerm _ =
  TyBool
```

Type system (maths)

$$\frac{}{\mathsf{true} : \mathsf{Bool}}\ \text{T-True}$$

$$\frac{}{\mathsf{false} : \mathsf{Bool}}\ \text{T-False}$$

$$\frac{t_1 : \mathsf{Bool} \quad t_2 : \mathsf{T} \quad t_3 : \mathsf{T}}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \mathsf{T}}\ \text{T-If}$$

```haskell
data TypeError =
    UnknownType
  | Unexpected { actual :: Type, expected :: Type }
  | ExpectedEq { type1 :: Type, type2 :: Type }
  deriving (Eq, Ord,Show)

expect :: MonadError TypeError m
       => Type
       -> Type
       -> m ()

expectEq :: MonadError TypeError m
       => Type
       -> Type
       -> m ()
```

$$\frac{}{\text{true} : \text{Bool}} \text{ T-True}$$

```
inferTmTrue :: Monad m
            => Term
            -> Maybe (m Type)
inferTmTrue TmTrue =
  Just $ return TyBool
inferTmTrue _ =
  Nothing
```

$$\frac{}{\mathsf{false} : \mathsf{Bool}}\ \textsf{T-False}$$

```
inferTmFalse :: Monad m
             => Term
             -> Maybe (m Type)
inferTmFalse TmFalse =
  Just $ return TyBool
inferTmFalse _ =
  Nothing
```

$$\frac{t_1 : \mathsf{Bool} \quad t_2 : \mathsf{T} \quad t_3 : \mathsf{T}}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \mathsf{T}}\ \text{T-If}$$

```haskell
inferTmIf :: MonadError TypeError m
          => (Term -> m Type)
          -> Term
          -> Maybe (m Type)
inferTmIf infer (TmIf tm1 tm2 tm3) = Just $ do




inferTmIf _ _ =
  Nothing
```

$$\frac{t_1 : \mathsf{Bool} \quad t_2 : \mathsf{T} \quad t_3 : \mathsf{T}}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \mathsf{T}}\ \text{T-If}$$

```
inferTmIf :: MonadError TypeError m
          => (Term -> m Type)
          -> Term
          -> Maybe (m Type)
inferTmIf infer (TmIf tm1 tm2 tm3) = Just $ do
  ty1 <- infer tm1
  expect ty1 TyBool




inferTmIf _ _ =
  Nothing
```

$$\frac{t_1 : \mathsf{Bool} \quad t_2 : \mathsf{T} \quad t_3 : \mathsf{T}}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \mathsf{T}}\ \textsf{T-If}$$

```haskell
inferTmIf :: MonadError TypeError m
          => (Term -> m Type)
          -> Term
          -> Maybe (m Type)
inferTmIf infer (TmIf tm1 tm2 tm3) = Just $ do
  ty1 <- infer tm1
  expect ty1 TyBool
  ty2 <- infer tm2
  ty3 <- infer tm3
  expectEq ty2 ty3

inferTmIf _ _ =
  Nothing
```

$$\frac{t_1 : \mathsf{Bool} \quad t_2 : \mathsf{T} \quad t_3 : \mathsf{T}}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : \mathsf{T}}\ \textsf{T-If}$$

```haskell
inferTmIf :: MonadError TypeError m
          => (Term -> m Type)
          -> Term
          -> Maybe (m Type)
inferTmIf infer (TmIf tm1 tm2 tm3) = Just $ do
  ty1 <- infer tm1
  expect ty1 TyBool
  ty2 <- infer tm2
  ty3 <- infer tm3
  expectEq ty2 ty3
  return ty2
inferTmIf _ _ =
  Nothing
```

```haskell
inferRules :: MonadError TypeError m
           => [Term -> Maybe (m Type)]
inferRules =
  [ inferTmTrue
  , inferTmFalse
  , inferTmIf infer
  ]

infer :: MonadError TypeError m
      => Term
      -> m Type
infer tm =
  fromMaybe (throwError UnknownType) .
  asum .
  fmap ($ tm) $
  inferRules
```

Properties of the type system

Progress: Well-typed terms are either values or can take a step.

Preservation: Well-typed terms do not change type when they take a step.

Aside: Not all ill-typed terms are stuck

```
> smallStep $ TmIf TmTrue TmFalse (TmInt 12)
TmFalse
```

Testing the type system

Need some more generators and friends

```
genAnyType        :: Gen Type
genNotType        :: Type -> Gen Type
genWellTypedTerm :: Type -> Gen Term
genIllTypedTerm  :: Type -> Gen Term
```

```haskell
newtype WellTypedTerm = WellTypedTerm Term

instance Arbitrary WellTypedTerm where
  arbitrary = genType >>= genWellTypedTerm
  shrink = shrinkWellTypedTerm
```

```haskell
prop_progress :: WellTypedTerm -> Property
prop_progress (WellTypedTerm tm) =
  isValue tm .||. canStep tm
```

```haskell
prop_preservation :: WellTypedTerm -> Property
prop_preservation (WellTypedTerm tm) =
  case smallStep tm of
    Nothing -> property True
    Just tm' -> infer tm === infer tm'
```

Good to double check that ill-typed terms result in type errors and that well-typed terms result in types.

```
prop_illTypedInfer :: IllTypedTerm -> Property
prop_illTypedInfer (IllTypedTerm tm ) =
  isLeft . runInfer . inferTerm $ tm

prop_wellTypedInfer :: WellTypedTerm -> Property
prop_wellTypedInfer (WellTypedTerm tm ) =
  isRight . runInfer . inferTerm $ tm
```

Parsing and printing B

Parse with `parsers` and `trifecta`, print with `ansi` $-$ `wl` $-$ `pprint`.

```haskell
parseTmFalse :: (Monad m , TokenParsing m)
             => m Term
parseTmFalse =
  TmFalse <$ symbol "False" <?> "False"

prettyTmFalse :: Term -> Maybe Doc
prettyTmFalse TmFalse =
  Just $ text "False"
prettyTmFalse _ =
  Nothing
```

```haskell
parseTmTrue :: (Monad m , TokenParsing m)
            => m Term
parseTmTrue =
  TmTrue <$ symbol "True" <?> "True"

prettyTmTrue :: Term -> Maybe Doc
prettyTmTrue TmTrue =
  Just $ text "True"
prettyTmTrue _ =
  Nothing
```

```haskell
parseTmIf :: (Monad m , TokenParsing m)
          => m Term -> m Term
parseTmIf parseTerm =
  TmIf <$
    symbol "if" <*> parseTerm <*
    symbol "then" <*> parseTerm <*
    symbol "else" <*> parseTerm
  <?> "if-then-else"
```

```haskell
prettyTmIf :: (Term -> Doc) -> Term -> Maybe Doc
prettyTmIf prettyTerm (TmIf tm1 tm2 tm3) =
  Just $
    text "if" <+> prettyTerm tm1 </>
    text "then" <+> prettyTerm tm2 </>
    text "else" <+> prettyTerm tm3
prettyTmIf _ _ =
  Nothing
```

```haskell
withParens :: TokenParsing m
           => m Term -> m Term
withParens p = parens p <|> p

parseTermRules :: (Monad m , TokenParsing m)
               => [m Term]
parseTermRules =
  [ parseTmFalse
  , parseTmTrue
  , parseTmIf parseTerm
  ]

parseTerm :: (Monad m , TokenParsing m)
          => m Term
parseTerm =
  ( withParens .
    asum $
    parseTermRules
  ) <?> "term"
```

```haskell
prettyTermRules :: [Term -> Maybe Doc]
prettyTermRules =
  [ prettyTmFalse
  , prettyTmTrue
  , prettyTmIf prettyTerm
  ]

prettyTerm :: Term -> Doc
prettyTerm tm =
  fromMaybe (text "???") .
  asum .
  fmap ($ tm) $
  prettyTermRules
```

Tip: Use the token parsers / identifier styles if you can

```
identifierStyle :: TokenParsing m
                => IdentifierStyle m
identifierStyle =
  IdentifierStyle {
    _styleName      = "identitfier"
  , _styleStart     = lower <|> char '_'
  , _styleLetter    = alphaNum <|> char '_'
  , _styleReserved  = HS.fromList
      ["if" , "then" , "else"]
  , _styleHighlight = Identifier
  , _styleReservedHighlight = ReservedIdentifier
  }

reservedIdentifier :: (Monad m , TokenParsing m)
                   => String -> m ()
reservedIdentifier =
  reserve identifierStyle
```

Tip: Steal the highlighting from `trifecta` for your printing.

```haskell
import Text.Parser.Token.Highlight  (Highlight (..))
import Text.Trifecta.Highlight      (withHighlight)

reservedIdentifier :: String -> Doc
reservedIdentifier =
  withHighlight ReservedIdentifier .
  text
```

```haskell
parseTmIf :: (Monad m , TokenParsing m)
          => m Term -> m Term
parseTmIf parseTerm =
  TmIf <$
    reservedIdentifier "if" <*> parseTerm <*
    reservedIdentifier "then" <*> parseTerm <*
    reservedIdentifier "else" <*> parseTerm
  <?> "if-then-else"
```

```haskell
prettyTmIf :: (Term -> Doc) -> Term -> Maybe Doc
prettyTmIf prettyTerm (TmIf tm1 tm2 tm3) =
  Just $
    reservedIdentifier "if" <+> prettyTerm tm1 </>
    reservedIdentifier "then" <+> prettyTerm tm2 </>
    reservedIdentifier "else" <+> prettyTerm tm3
prettyTmIf _ _ =
  Nothing
```

Tip: At least do minimal testing of your parser and printer

```
prop_prettyParse :: AnyTerm -> Property
prop_prettyParse (AnyTerm tm) =
  case (parseTermString . prettyTermString) tm of
    Left _ -> property False
    Right tm' -> tm === tm'
```

Now we have the core of a REPL

```
parseAndEval :: String
             -> Doc
parseAndEval s =
```

Now we have the core of a REPL

```
parseAndEval :: String
             -> Doc
parseAndEval s =
  case parseFromString parseTerm s of
    Left d -> d
```

Now we have the core of a REPL

```
parseAndEval :: String
             -> Doc
parseAndEval s =
  case parseFromString parseTerm s of
    Left d -> d
    Right tm -> case runInfer . inferTerm $ tm of
```

Now we have the core of a REPL

```
parseAndEval :: String
             -> Doc
parseAndEval s =
  case parseFromString parseTerm s of
    Left d -> d
    Right tm -> case runInfer . inferTerm $ tm of
      Left e ->
        prettyTypeError e
```

Now we have the core of a REPL

```
parseAndEval :: String
             -> Doc
parseAndEval s =
  case parseFromString parseTerm s of
    Left d -> d
    Right tm -> case runInfer . inferTerm $ tm of
      Left e ->
        prettyTypeError e
      Right ty ->
        prettyTerm (smallStepEval tm) <+>
        text ":" <+>
        prettyType ty
```

B with Class

```haskell
data BoolTerm tm =
    TmFalse
  | TmTrue
  | TmIf tm tm tm

makeClassyPrisms ''BoolTerm
```

```haskell
class AsBoolTerm s tm | s -> tm where
  _BoolTerm :: Prism' s (BoolTerm tm)
  _TmFalse  :: Prism' s ()
  _TmTrue   :: Prism' s ()
  _TmIf     :: Prism' s (tm, tm, tm)

  _TmFalse = _BoolTerm . _TmFalse
  _TmTrue  = _BoolTerm . _TmTrue
  _TmIf    = _BoolTerm . _TmIf

instance AsBoolTerm (BoolTerm tm) tm
  _BoolTerm = id
  _TmFalse = ...
  _TmTrue = ...
  _TmIf = ...
```

```haskell
data MyTerm =
    MyBoolTerm (BoolTerm MyTerm)
  | MyNatTerm  (NatTerm MyTerm)

makeClassyPrisms ''MyTerm

instance AsBoolTerm MyTerm MyTerm where
  _BoolTerm = _MyBoolTerm
```

```
> :t preview _TmFalse
AsBoolTerm tm => tm -> Maybe ()

> preview _TmFalse TmFalse
Just ()

> preview _TmFalse (MyBoolTerm TmFalse)
Just ()

> preview _TmFalse (TmIf TmTrue TmFalse TmTrue)
Nothing
```

```
> :t review _TmFalse
AsBoolTerm tm => () -> tm

> (review _TmFalse ()) :: BoolTerm
TmFalse

> (review _TmFalse ()) :: MyTerm
MyBoolTerm TmFalse
```

```haskell
data BoolType =
  TyBool
  deriving (Eq, Ord, Show)

makeClassyPrisms ''BoolType
```

```haskell
eIfTrue :: AsBoolTerm tm tm
        => tm
        -> Maybe tm
eIfTrue tm = do
  (tm1, tm2, _) <- preview _TmIf tm
  _ <- preview _TmTrue tm1
  return tm2
```

```haskell
eIf :: AsBoolTerm tm tm
    => (tm -> Maybe tm)
    -> tm
    -> Maybe tm
eIf step tm = do
  (tm1, tm2, tm3) <- preview _TmIf tm
  tm1' <- step tm1
  return $ review _TmIf (tm1', tm2, tm3)
```

```haskell
inferTmIf :: ( Eq ty
             , AsUnexpected e ty , AsExpectedEq e ty
             , MonadError e m
             , AsBoolTerm tm tm , AsBoolType ty ty
             )
          => (tm -> m ty)
          -> tm
          -> Maybe (m ty)
inferTmIf infer tm = do
  (tm1, tm2, tm3) <- preview _TmIf tm
  return $ do
    ty1 <- infer tm1
    expect ty1 (review _TyBool ())
    ty2 <- infer tm2
    ty3 <- infer tm3
    expectEq ty2 ty3
    return ty3
```

Data types for the various rules, monoids for all of the data types.

Annotating the AST

```
> if S O then False else True
```

```
Unexpected type:
  actual: Nat
  expected: Bool
```

```
> if S O then False else True
if S O then False else True<EOF>
   ~~~
Unexpected type:
  actual: Nat
  expected: Bool
```

```haskell
data NoteTerm n tm =
  TmNote n tm

makeClassyPrisms ''NoteTerm

data NoteType n ty =
  TyNote n ty

makeClassyPrisms ''NoteType
```

```haskell
withSpan :: ( Monad m
            , DeltaParsing m
            , AsNoteTerm tm Span tm
            )
         => m tm
         -> m tm
withSpan p = do
  (tm :~ s) <- spanned p
  return $ review _TmNote (s, tm)
```

```haskell
inferTmNote :: ( AsNoteType ty n ty
               , AsNoteTerm tm n tm
               , Monad m
               )
             => (tm -> m ty)
             -> tm
             -> Maybe (m ty)
inferTmNote infer tm = do
  (n, tm) <- preview _TmNote tm
  return $ do
    ty <- infer tm
    return $ review _TyNote (n, ty)
```

```
prettyUnexpected :: (ty -> Doc)
                 -> (ty, ty)
                 -> Doc
prettyUnexpected prettyType (ac, ex) =
  hang 2 (text "Unexpected type:" PP.<$>
          text "actual:" <+> prettyType ac PP.<$>
          text "expected:" <+> prettyType ex)
```

```haskell
prettyUnexpectedSrcLoc :: ( Show ty
                          , Renderable n
                          )
                       => (ty -> Doc)
                       -> (n, ty)
                       -> ty
                       -> Doc
prettyUnexpectedSrcLoc prettyType (n, ac) ex =
  (render n) PP.<$>
  hang 2 (text "Unexpected type:" PP.<$>
          text "actual:" <+> prettyType ac PP.<$>
          text "expected:" <+> prettyType ex)
```

STLC

Terms

$$
\begin{aligned}
t = {} & \mathsf{x} && \text{(Var)} \\
\mid {} & \lambda x\colon T.\, t && \text{(Lam)} \\
\mid {} & \mathsf{t\ t} && \text{(App)}
\end{aligned}
$$

Var

```
x + 1
```

Lam

```
\x : Int . x + 1
```

App

```
(\x : Int . x + 1) 2
```

Variables can be free or bound.

```
\x : Int . x + y
```

Variables can be free or bound.

```
x + (\x : Int . x * x) (x + 1)
```

Substitution can be simple.

```
(\x : Int . x + y) 2
            2 + y
```

Substitution can be tricky.

```
(\x : Int . x + (\x : Int . x * x) (x + 1)) 2
            2 + (\x : Int . x * x) (2 + 1)
            2 + (\x : Int . x * x) 3
            2 +             3 * 3
```

Monads do substitution!

`bound` does the bookkeeping for us.

```haskell
data Var b a =
    B b
  | F a

newtype Scope b f a = ...
-- just don't look
```

```haskell
abstract1 :: (Monad f, Eq a)
          => a
          -> f    a
          -> Scope () f    a
```

```
abstract1 ::
           String
        -> Term String
        -> Scope () Term String
```

```haskell
abstract1 ::
             String
          -> Term String
          -> Scope () Term String

> abstract1
    "x"
    (TmIf (TmVar "x") TmFalse TmTrue)
```

```haskell
abstract1 ::
             String
          -> Term String
          -> Scope () Term String

> abstract1
    "x"
    (TmIf (TmVar "x") TmFalse TmTrue)

Scope (TmIf (TmVar (B ())) TmFalse TmTrue)
```

```haskell
instantiate1 :: Monad f
             => f    a
             -> Scope () f    a
             -> f    a
```

```haskell
instantiate1 ::
           Term String
        -> Scope () Term String
        -> Term String
```

```haskell
instantiate1 ::
                Term String
             -> Scope () Term String
             -> Term String

> instantiate1
    (pure "x")
    (Scope (TmIf (TmVar (B ())) TmFalse TmTrue))
```

```
instantiate1 ::
                Term String
             -> Scope () Term String
             -> Term String

> instantiate1
    (pure "x")
    (Scope (TmIf (TmVar (B ())) TmFalse TmTrue))

TmIf (TmVar "x") TmFalse TmTrue
```

```haskell
instantiate1 ::
                Term String
             -> Scope () Term String
             -> Term String

> instantiate1
    TmTrue
    (Scope (TmIf (TmVar (B ())) TmFalse TmTrue))
```

```
instantiate1 ::
               Term String
            -> Scope () Term String
            -> Term String

> instantiate1
    TmTrue
    (Scope (TmIf (TmVar (B ())) TmFalse TmTrue))

TmIf TmTrue TmFalse TmTrue
```

```
abstract :: Monad f
         => (a      -> Maybe b  )
         -> f    a
         -> Scope b   f    a
```

```
abstract ::
      (String -> Maybe Int)
   -> Term String
   -> Scope Int Term String
```

```haskell
abstract ::
           (String -> Maybe Int)
        -> Term String
        -> Scope Int Term String

> abstract
    (‘elemIndex‘ ["x", "y"])
    (TmIf TmTrue (TmVar "x") (TmVar "y"))
```

```haskell
abstract ::
           (String -> Maybe Int)
        -> Term String
        -> Scope Int Term String

> abstract
    (`elemIndex` ["x", "y"])
    (TmIf TmTrue (TmVar "x") (TmVar "y"))

Scope (TmIf TmTrue (TmVar (B 0)) (TmVar (B 1)))
```

```
abstract ::
            (String -> Maybe Int)
         -> Term String
         -> Scope Int Term String

> abstract
    (`elemIndex` ["x", "y"])
    (TmIf TmTrue (TmVar "x") (TmVar "z"))
```

```haskell
abstract ::
            (String -> Maybe Int)
         -> Term String
         -> Scope Int Term String

> abstract
    (`elemIndex` ["x", "y"])
    (TmIf TmTrue (TmVar "x") (TmVar "z"))

Scope (TmIf TmTrue (TmVar (B 0)) (TmVar (F (TmVar "z"))))
```

```
instantiate :: Monad f
            => (b   -> f    a)
            -> Scope b   f    a
            -> f    a
```

```haskell
instantiate ::
            (Int -> Term String)
         -> Scope Int Term String
         -> Term String
```

```haskell
instantiate ::
               (Int -> Term String)
            -> Scope Int Term String
            -> Term String

> instantiate
  ([TmFalse, TmTrue] !!)
  (Scope (TmIf TmTrue (TmVar (B 0)) (TmVar (B 1))))
```

```haskell
instantiate ::
               (Int -> Term String)
            -> Scope Int Term String
            -> Term String

> instantiate
  ([TmFalse, TmTrue] !!)
  (Scope (TmIf TmTrue (TmVar (B 0)) (TmVar (B 1))))

TmIf TmTrue TmFalse TmTrue
```

```
instantiate ::
                  (Int -> Term String)
              -> Scope Int Term String
              -> Term String

> instantiate
  ([TmFalse, TmTrue] !!)
  (Scope (TmIf TmTrue (TmVar (B 0)) (TmVar (F (TmVar "z")))
```

```haskell
instantiate ::
               (Int -> Term String)
             -> Scope Int Term String
             -> Term String

> instantiate
  ([TmFalse, TmTrue] !!)
  (Scope (TmIf TmTrue (TmVar (B 0)) (TmVar (F (TmVar "z")))

TmIf TmTrue TmFalse (TmVar "z")
```

```haskell
data StlcType ty =
    TyArr ty ty
  deriving (Eq, Ord, Show)

makeClassyPrisms ''StlcTerm
```

```haskell
data StlcTerm ty tm a =
    TmVar a
  | TmLam String ty (Scope () tm a)
  | TmApp tm tm
  deriving (Eq, Ord, Show, Functor, Foldable, Traversable)

makeClassyPrisms ''StlcTerm
```

```haskell
instance Eq1 (StlcTerm ty) where
  (==#) = (==)

instance Ord1 (StlcTerm ty) where
  compare1 = compare

instance Show1 (StlcTerm ty) where
  showsPrec1 = showsPrec
```

There is a typeclass `Bound` that provides $>>>=$, which helps do substitution "through" other structures.

```haskell
instance Monad tm => Monad (StlcTerm ty tm) where
  return = TmVar

  TmVar x       >>= f = f x

  TmLam v t s   >>= f = TmLam v t (s >>>= f)
  TmApp tm1 tm2 >>= f = TmApp (tm1 >>= f) (tm2 >>= f)
```

```haskell
data StlcVar a =
    TmVar a
  deriving (Eq, Ord, Show, Functor, Foldable, Traversable)
makeClassyPrisms ''StlcVar

data StlcTerm ty tm a =
    TmLam String ty (Scope () tm a)
  | TmApp tm tm
  deriving (Eq, Ord, Show, Functor, Foldable, Traversable)
makeClassyPrisms ''StlcTerm

instance Bound (StlcTerm ty) where
  TmLam v t s   >>>= f = TmLam v t (s >>>= f)
  TmApp tm1 tm2 >>>= f = TmApp (tm1 >>= f) (tm2 >>= f)
```

```haskell
data MyTerm a =
    TmVar a
    TmStlc (StlcTerm MyType (MyTerm a) a)
  | TmBool (BoolTerm (MyTerm a))
  deriving (Eq, Ord, Show, Functor, Foldable, Traversable)

instance Monad MyTerm where
  return          = TmVar

  TmVar x   >>= f = f x
  TmStlc tm >>= f = TmStlc (tm >>>= f)
  TmBool tm >>= f = TmBool (tm >>>= f)
```

Values

$$v = \lambda x \colon T.\, t$$

Small-step semantics

$$\frac{t_1 \longrightarrow {t_1}'}{t_1\, t_2 \longrightarrow {t_1}' t_2}\ \text{E-App1}$$

$$\frac{t_2 \longrightarrow {t_2}'}{v_1\, t_2 \longrightarrow v_1\, {t_2}'}\ \text{E-App1}$$

$$\frac{}{(\lambda x\colon\ T_{11}.\, t_{12}) v_2 \longrightarrow [x \mapsto v_2] t_{12}}\ \text{T-AppAbs}$$

$$\frac{}{(\lambda x\colon T_{11}.\, t_{12})v_2 \longrightarrow [x \mapsto v_2]t_{12}}\ \text{T-AppAbs}$$

```haskell
eAppAbs :: ( AsSTLCTerm tm ty tm a
           , Monad tm
           )
        => tm a
        -> Maybe (tm a)
eAppAbs tm = do
  (tm1, tm2) <- preview _TmApp tm
  (_, _, s) <- preview _TmLam tm1
  return $ instantiate1 tm2 s
```

Typing rules

$$\frac{x\colon T \in \Gamma}{\Gamma \vdash x\colon T}\ \text{T-Var}$$

$$\frac{\Gamma, x_1\colon T_1 \vdash t_2\colon T_2}{\Gamma \vdash \lambda x_1\colon T_1.\, t_2\colon T_1 \to T_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash t_1\colon T_1 \to T_2 \quad \Gamma \vdash t_2\colon T_1}{t_1\, t_2\colon T_2}\ \text{T-App}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}$$

```
inferTmVar :: (







             )
           => tm a
           -> Maybe (m ty)
inferTmVar tm = do
```

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsf{T-Var}$$

```
inferTmVar :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm a




              )
           => tm a
           -> Maybe (m ty)
inferTmVar tm = do
```

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}$$

```
inferTmVar :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm a
              , Ord a
              , MonadReader (M.Map a ty) m


              )
           => tm a
           -> Maybe (m ty)
inferTmVar tm = do
```

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var}$$

```haskell
inferTmVar :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm a
              , Ord a
              , MonadReader (M.Map a ty) m
              , AsFreeVar e a
              , MonadError e m
              )
           => tm a
           -> Maybe (m ty)
inferTmVar tm = do
```

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}$$

```haskell
inferTmVar :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm a
              , Ord a
              , MonadReader (M.Map a ty) m
              , AsFreeVar e a
              , MonadError e m
              )
           => tm a
           -> Maybe (m ty)
inferTmVar tm = do
  v <- preview _TmVar tm
```

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}$$

```
inferTmVar :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm a
              , Ord a
              , MonadReader (M.Map a ty) m
              , AsFreeVar e a
              , MonadError e m
              )
           => tm a
           -> Maybe (m ty)
inferTmVar tm = do
  v <- preview _TmVar tm
  case asks (M.lookup v) of
```

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var}$$

```
inferTmVar :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm a
              , Ord a
              , MonadReader (M.Map a ty) m
              , AsFreeVar e a
              , MonadError e m
              )
           => tm a
           -> Maybe (m ty)
inferTmVar tm = do
  v <- preview _TmVar tm
  case asks (M.lookup v) of
    Nothing -> throwing _FreeVar a
```

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ T-Var}$$

```haskell
inferTmVar :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm a
              , Ord a
              , MonadReader (M.Map a ty) m
              , AsFreeVar e a
              , MonadError e m
              )
           => tm a
           -> Maybe (m ty)
inferTmVar tm = do
  v <- preview _TmVar tm
  case asks (M.lookup v) of
    Nothing -> throwing _FreeVar a
    Just ty -> return ty
```

$$\frac{\Gamma, x_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x_1 : T_1 .\, t_2 : T_1 \to T_2} \text{ T-Abs}$$

```
inferTmLam :: (




               )
           => (tm String -> m ty)
           -> tm String
           -> Maybe (m ty)
inferTmLam infer tm = do
```

$$\frac{\Gamma, x_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x_1 : T_1 . t_2 : T_1 \rightarrow T_2} \text{T-Abs}$$

```
inferTmLam :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm String



              )
           => (tm String -> m ty)
           -> tm String
           -> Maybe (m ty)
inferTmLam infer tm = do
```

$$\frac{\Gamma, x_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x_1 : T_1 . t_2 : T_1 \to T_2} \text{T-Abs}$$

```
inferTmLam :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm String
              , MonadReader (M.Map String ty) m

              )
           => (tm String -> m ty)
           -> tm String
           -> Maybe (m ty)
inferTmLam infer tm = do
```

$$\frac{\Gamma, x_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x_1 : T_1 .\, t_2 : T_1 \to T_2} \text{ T-Abs}$$

```
inferTmLam :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm String
              , MonadReader (M.Map String ty) m
              , Monad tm
              )
           => (tm String -> m ty)
           -> tm String
           -> Maybe (m ty)
inferTmLam infer tm = do
```

$$\frac{\Gamma, x_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x_1 : T_1 .\, t_2 : T_1 \to T_2} \text{ T-Abs}$$

```haskell
inferTmLam :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm String
              , MonadReader (M.Map String ty) m
              , Monad tm
              )
           => (tm String -> m ty)
           -> tm String
           -> Maybe (m ty)
inferTmLam infer tm = do
  (n, ty1, s) <- preview _TmLam
```

$$\frac{\Gamma, x_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x_1 : T_1 .\, t_2 : T_1 \to T_2} \text{ T-Abs}$$

```
inferTmLam :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm String
              , MonadReader (M.Map String ty) m
              , Monad tm
              )
           => (tm String -> m ty)
           -> tm String
           -> Maybe (m ty)
inferTmLam infer tm = do
  (n, ty1, s) <- preview _TmLam
  ty2 <- local (M.insert n ty1) $
    infer (                                 )
```

$$\frac{\Gamma, x_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x_1 : T_1 . t_2 : T_1 \to T_2} \text{T-Abs}$$

```
inferTmLam :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm String
              , MonadReader (M.Map String ty) m
              , Monad tm
              )
           => (tm String -> m ty)
           -> tm String
           -> Maybe (m ty)
inferTmLam infer tm = do
  (n, ty1, s) <- preview _TmLam
  ty2 <- local (M.insert n ty1) $
    infer (instantiate1 (review _TmVar n) s)
```

$$\frac{\Gamma, x_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x_1 : T_1 .\, t_2 : T_1 \to T_2}\ \text{T-Abs}$$

```haskell
inferTmLam :: ( AsSTLCType ty ty
              , AsSTLCTerm tm ty tm String
              , MonadReader (M.Map String ty) m
              , Monad tm
              )
           => (tm String -> m ty)
           -> tm String
           -> Maybe (m ty)
inferTmLam infer tm = do
  (n, ty1, s) <- preview _TmLam
  ty2 <- local (M.insert n ty1) $
    infer (instantiate1 (review _TmVar n) s)
  return $ review _TyArr (ty1, ty2)
```

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{t_1 t_2 : T_2}\ \text{T-App}$$

```
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a



              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
```

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{t_1 t_2 : T_2} \text{ T-App}$$

```haskell
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a
              , AsUnexpected e ty, AsNotArrow e ty, Eq ty
              , MonadError e m
              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
```

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{t_1\, t_2 : T_2}\ \text{T-App}$$

```
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a
              , AsUnexpected e ty, AsNotArrow e ty, Eq ty
              , MonadError e m
              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
```

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{t_1\, t_2 : T_2} \text{ T-App}$$

```haskell
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a
              , AsUnexpected e ty, AsNotArrow e ty, Eq ty
              , MonadError e m
              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
    ty1 <- infer tm1
    ty2 <- infer tm2
```

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{t_1\, t_2 : T_2} \text{ T-App}$$

```
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a
              , AsUnexpected e ty, AsNotArrow e ty, Eq ty
              , MonadError e m
              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
    ty1 <- infer tm1
    ty2 <- infer tm2
    case preview _TyArr ty1 of
```

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{t_1 \, t_2 : T_2} \text{T-App}$$

```
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a
              , AsUnexpected e ty, AsNotArrow e ty, Eq ty
              , MonadError e m
              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
    ty1 <- infer tm1
    ty2 <- infer tm2
    case preview _TyArr ty1 of
      Nothing -> throwing _NotArrow (ty1, ty2)
```

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{t_1\, t_2 : T_2}\ \text{T-App}$$

```
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a
              , AsUnexpected e ty, AsNotArrow e ty, Eq ty
              , MonadError e m
              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
    ty1 <- infer tm1
    ty2 <- infer tm2
    case preview _TyArr ty1 of
      Nothing -> throwing _NotArrow (ty1, ty2)
      Just (tyFrom, tyTo) -> do
```

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{t_1 t_2 : T_2}\ \textsf{T-App}$$

```
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a
              , AsUnexpected e ty, AsNotArrow e ty, Eq ty
              , MonadError e m
              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
    ty1 <- infer tm1
    ty2 <- infer tm2
    case preview _TyArr ty1 of
      Nothing -> throwing _NotArrow (ty1, ty2)
      Just (tyFrom, tyTo) -> do
        expect tyFrom ty2
```

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{t_1\, t_2 : T_2}\ \text{T-App}$$

```haskell
inferTmApp :: ( AsSTLCType ty ty, AsSTLCTerm tm ty tm a
              , AsUnexpected e ty, AsNotArrow e ty, Eq ty
              , MonadError e m
              )
           => (tm a -> m ty) -> tm a -> Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
    ty1 <- infer tm1
    ty2 <- infer tm2
    case preview _TyArr ty1 of
      Nothing -> throwing _NotArrow (ty1, ty2)
      Just (tyFrom, tyTo) -> do
        expect tyFrom ty2
        return tyTo
```

Extending B (part 2)

```
(\not : Bool -> Bool =>
  (\x : Bool =>
    if x then False else True
  )
) (... code using not ...)
```

```
let
  not x = if x then False else True
in
  ... code using not ...
```

```
import Prelude (not)

... code using not ...
```

Conclusion

Thanks!

All of this and more will appear on dlaing.org before too long.

Further avenues

Many more pieces we can add

Pattern matching gets fun (and can use bound)

ADTs gets fun (and can use bound)

Type inference and/or bidirectional typing

Tagged and/or hbound

LLVM output