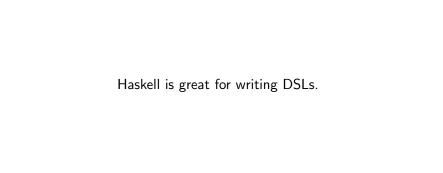
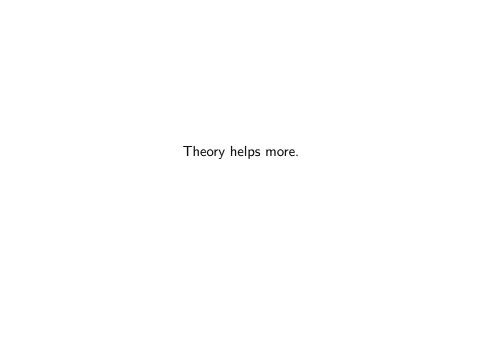
## Little Languages

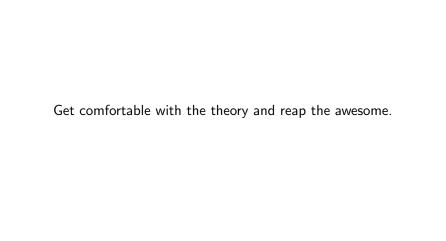
Dave Laing

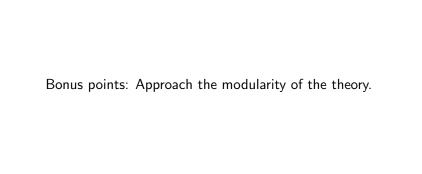
YOW! Lambda Jam 2016



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







В

## Terms (maths)

| if t then t else t

.

 $t = \mathsf{true}$  | false

### 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

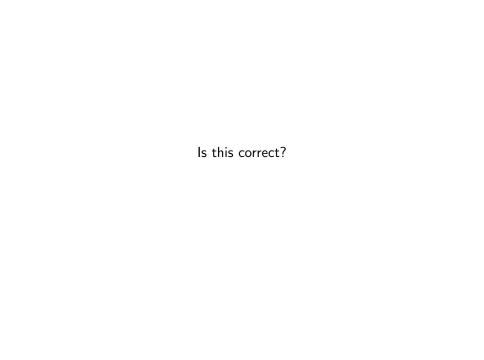
```
data Term =
    ...
    | TmNot Term
    | TmAnd Term Term
    | TmOr Term Term
```

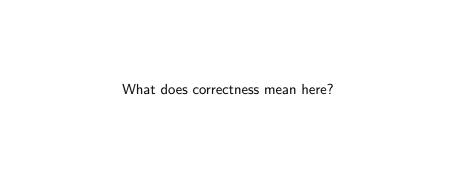
#### An evaluator (?)

```
eval :: Term
    -> Term

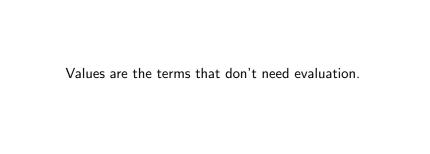
eval TmTrue = TmTrue

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





## Values



## Values (maths)

```
v = \mathsf{true}
| false
```

## Values (maths)

false val V-False

true val

#### Values (Haskell)

#### Values (Haskell)

# true val V-True

# false val V-False

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

```
import Data.Foldable (asum)
```

```
valueRules :: [Term -> Maybe Term]
valueRules =
  [ valueTmTrue
  , valueTmFalse
value :: Term
      -> Maybe Term
value tm =
  asum .
  fmap ($ tm) $
```

valueRules



## Small-step semantics (maths)

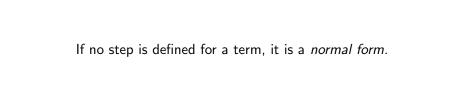
if true then 
$$t_2$$
 else  $t_3 \longrightarrow t_2$  E-IfTrue

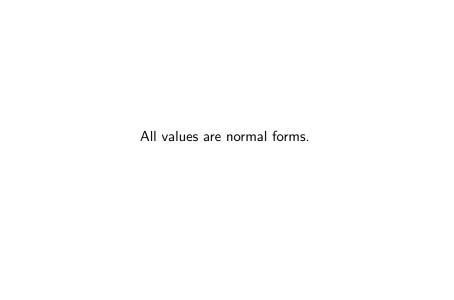
if true then 
$$t_2$$
 else  $t_3 \longrightarrow t_2$ 

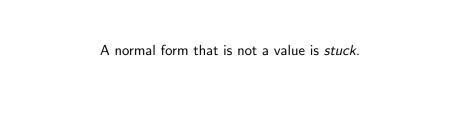
if false then  $t_2$  else  $t_3 \longrightarrow t_3$ 

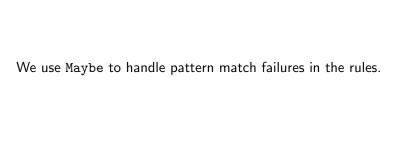
E-IfFalse

 $rac{t_1 \longrightarrow t_1'}{ ext{if } t_1 ext{ then } t_2 ext{ else } t_3 \longrightarrow ext{if } t_1' ext{ then } t_2 ext{ else } t_3}$  E-If









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

```
\overline{ \text{ if true then } t_2 \text{ else } t_3 \longrightarrow t_2 } \text{ E-IfTrue} \\
```

```
-> Maybe Term
eIfTrue (TmIf TmTrue tm2 _) =
```

eIfTrue (TmIf TmTrue tm2 \_) =
 Just tm2

eIfTrue \_ =
 Nothing

eIfTrue :: 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</pre>

eIf =

Nothing

```
smallStepRules :: [Term -> Maybe Term]
smallStepRules =
  [ eIfTrue
  , eIfFalse
  , eIf smallStep
smallStep :: Term
          -> Maybe Term
smallStep tm =
  asum .
  fmap ($ tm) $
  smallStepRules
```

### An evaluator (!)

Properties of the small-step semantics

There are	me properties that we want to hold for our sma function.	ll-step

All values are normal forms.

No terms are stuck.

# Small-step is determinate.

Property 3

Small-step is normalizing.

Property 4

# Testing the small-step semantics

### Need some generators and friends

genAnyTerm :: Gen Term

shrAnyTerm :: Term -> [Term]

### We can wrap them in newtypes

```
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 -> ...

isNothing . smallStep

isNormalForm =

```
We also need some helpers
termSize TmFalse =
   1
termSize TmTrue =
   1
termSize (TmIf tm1 tm2 tm3) =
   1 + termSize tm1 + termSize tm2 + termSize tm3
```

All values are normal forms.

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

### No terms are stuck.

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

### Small-step is determinate.

```
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
```

## Small-step is normalizing.

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

# Bonus Property

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



data Type =
 TyBool

Type system (maths)

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

```
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 ()
```

```
true: Bool T-True
```

# false: Bool T-False

```
\frac{t_1 \colon \mathsf{Bool} \ t_2 \colon \mathsf{T} \ t_3 \colon \mathsf{T}}{\mathsf{if} \ t_1 \ \mathsf{then} \ t_2 \ \mathsf{else} \ t_3 \colon \mathsf{T}} \mathsf{T-lf}
\mathsf{inferTmIf} \ \colon \colon \mathsf{MonadError} \ \mathsf{TypeError} \ \mathsf{m}
= > \ (\mathsf{Term} \ -> \ \mathsf{m} \ \mathsf{Type})
-> \ \mathsf{Term}
-> \ \mathsf{Maybe} \ (\mathsf{m} \ \mathsf{Type})
\mathsf{inferTmIf} \ \mathsf{infer} \ (\mathsf{TmIf} \ \mathsf{tm1} \ \mathsf{tm2} \ \mathsf{tm3}) \ = \ \mathsf{Just} \ \$ \ \mathsf{do}
```

```
inferTmIf _ _ =
Nothing
```

```
\frac{t_1 \colon \mathsf{Bool} \ t_2 \colon \mathsf{T} \ t_3 \colon \mathsf{T}}{\mathsf{if} \ t_1 \ \mathsf{then} \ t_2 \ \mathsf{else} \ t_3 \colon \mathsf{T}} \, \mathsf{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
```

```
t_1: Bool t_2: T t_3: T t_3: T 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
 ty2 <- infer tm2
 ty3 <- infer tm3
  expectEq ty2 ty3
inferTmIf _ =
 Nothing
```

```
t_1: Bool t_2: T t_3: T t_3: T 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
 ty2 <- infer tm2
 ty3 <- infer tm3
  expectEq ty2 ty3
 return ty2
inferTmIf _ =
  Nothing
```

```
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
```

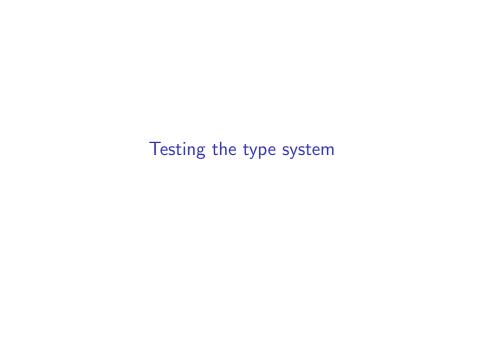


Progress: Well-typed terms are either values or can take a step.

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



# Need some more generators and friends

 $\begin{array}{lll} \mbox{\tt genAnyType} & :: \mbox{\tt Gen Type} \\ \mbox{\tt genNotType} & :: \mbox{\tt Type} & -> \mbox{\tt Gen Type} \end{array}$ 

 ${\tt genWellTypedTerm} \ :: \ {\tt Type} \ {\tt ->} \ {\tt Gen} \ {\tt Term}$ 

genIllTypedTerm :: Type -> Gen Term

newtype WellTypedTerm = WellTypedTerm Term

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

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

```
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
```



Parse with parsers and trifecta, print with

 $\verb"ansi-wl-pprint".$ 

<?> "if-then-else"

```
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

```
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"
```

```
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.

import Text.Parser.Token.Highlight (Highlight (...))

import Text.Trifecta.Highlight (withHighlight)
```

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

```
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

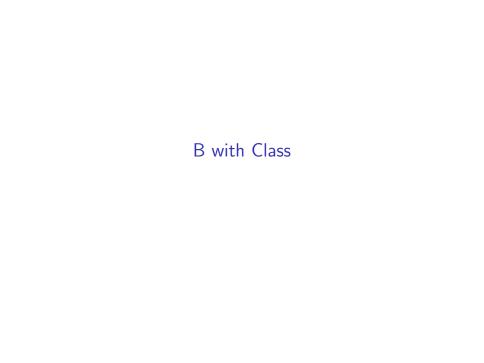
prop\_prettyParse :: AnyTerm -> Property

prop\_prettyParse (AnyTerm tm) =

case (parseTermString . prettyTermString) tm of

Left \_ -> property False
Right tm' -> tm === tm'

```
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
```



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

makeClassyPrisms ''BoolTerm
```

```
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
 _{\rm TmFalse} = \dots
 _TmTrue = ...
```

TmIf = ...

```
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

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

makeClassyPrisms ''BoolType

```
eIf :: AsBoolTerm tm tm
=> (tm -> Maybe tm)
-> tm
-> Maybe tm
```

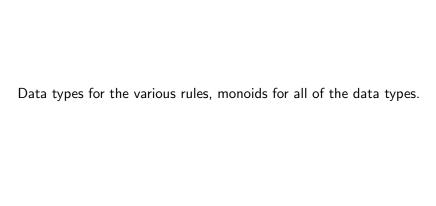
(tm1, tm2, tm3) <- preview \_TmIf tm

return \$ review \_TmIf (tm1', tm2, tm3)

eIf step tm = do

tm1' <- step tm1

```
inferTmIf :: ( Eq ty
             , AsUnexpected e ty , AsExpectedEq e ty
             , MonadError e m
             , AsBoolTerm tm tm , AsBoolType ty ty
          => (tm -> m ty)
          -> t.m
          -> Maybe (m ty)
inferTmIf infer tm = do
  (tm1, tm2, tm3) <- preview _TmIf tm
 return $ do
   tv1 <- infer tm1
    expect ty1 (review _TyBool ())
   ty2 <- infer tm2
   ty3 <- infer tm3
   expectEq ty2 ty3
   return ty3
```





> 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

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

makeClassyPrisms ''NoteTerm

```
data NoteType n ty =
  TyNote n ty
```

makeClassyPrisms ''NoteType

(n, tm) <- preview \_TmNote tm

return \$ review \_TyNote (n, ty)

return \$ do

ty <- infer tm

```
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)
```



# Terms

$$egin{array}{ll} t=\mathsf{x} & (\mathsf{Var}) \ &|\; \lambda x\colon T\ldotp t & (\mathsf{Lam}) \ &|\; \mathsf{t} \;\mathsf{t} & (\mathsf{App}) \end{array}$$

# 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
```

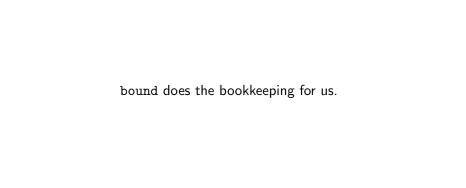
# Substitution can be tricky.

3 \* 3

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

2. +





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

newtype Scope b f a = ...
```

-- just don't look

#### abstract1 ::

- String
- -> Term String
- -> Scope () Term String

(TmIf (TmVar "x") TmFalse TmTrue)

 $^{II}$  $^{X}$  $^{II}$ 

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

```
instantiate1 :: Monad f
           => f a
```

-> Scope () f a -> f a

#### instantiate1 ::

- Term String
- -> Scope () Term String
- -> Term String

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

TmIf (TmVar "x") TmFalse TmTrue

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

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

> instantiate1

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

TmIf TmTrue TmFalse TmTrue

#### abstract ::

- (String -> Maybe Int)
- -> Term String
- -> Scope Int Term String

(TmIf TmTrue (TmVar "x") (TmVar "y"))

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

(TmIf TmTrue (TmVar "x") (TmVar "z"))

> abstract

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

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

### instantiate ::

- (Int -> Term String)
- -> Scope Int Term String
- -> Term String

TmIf TmTrue TmFalse TmTrue

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

> instantiate

([TmFalse, TmTrue] !!)

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

TmIf TmTrue TmFalse (TmVar "z")

([TmFalse, TmTrue] !!)

> instantiate

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

makeClassyPrisms ''StlcTerm

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

```
instance Eq1 (StlcTerm ty) where
  (==#) = (==)
instance Ord1 (StlcTerm ty) where
```

compare1 = compare

showsPrec1 = showsPrec

instance Show1 (StlcTerm ty) where

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

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)

 $\label{eq:tmapp} \mbox{TmApp tm1 tm2} >>= \mbox{f} = \mbox{TmApp (tm1 >>= f) (tm2 >>= f)}$ 

```
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)

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

makeClassyPrisms ''StlcTerm

```
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

$$v = \lambda x$$
:  $T$ .  $t$ 

## Small-step semantics

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

 $rac{t_2 \longrightarrow {t_2}'}{v_1 t_2 \longrightarrow {v_1} {t_2}'}$ E-App1

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

, Monad tm

eAppAbs :: ( AsSTLCTerm tm ty tm a

=> tm a

eAppAbs tm = do

-> Maybe (tm a)

(tm1, tm2) <- preview \_TmApp tm (\_, \_, s) <- preview \_TmLam tm1 return \$ instantiate1 tm2 s

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

Typing rules

$$x: T \in \Gamma$$

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

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

$$\frac{\Gamma, x_1 \colon T_1 \vdash t_2 \colon T_2}{\Gamma \vdash \lambda x_1 \colon T_1 \colon t_2 \colon T_1 \to T_2}$$

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

```
\frac{x \colon T \in \Gamma}{\Gamma \vdash x \colon T} \mathsf{T-Var}
```

inferTmVar :: (

```
\frac{x \colon T \in \Gamma}{\Gamma \vdash x \colon T} \mathsf{T-Var}
```

inferTmVar tm = do

```
\frac{x \colon T \in \Gamma}{\Gamma \vdash x \colon T} \mathsf{T-Var}
```

-> Maybe (m ty)

inferTmVar tm = do

```
\frac{x \colon T \in \Gamma}{\Gamma \vdash x \colon T} \mathsf{T-Var}
```

-> Maybe (m ty)

inferTmVar tm = do

```
\frac{x\colon T\in\Gamma}{\Gamma\vdash x\colon T}\,\mathsf{T\text{-}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

v <- preview \_TmVar tm

inferTmVar tm = do

-> Maybe (m ty)

```
\frac{x\colon T\in \Gamma}{\Gamma\vdash x\colon T}\operatorname{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</pre>
```

```
x: T \in \Gamma
\Gamma \vdash x: T 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

```
x: T \in \Gamma
\Gamma \vdash x: T 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
    Just ty -> return ty
```

```
\frac{\Gamma, x_1 \colon T_1 \vdash t_2 \colon T_2}{\Gamma \vdash \lambda x_1 \colon T_1 \colon t_2 \colon T_1 \to T_2} \text{ T-Abs} inferTmLam :: (
```

-> tm String
-> Maybe (m ty)

inferTmLam infer tm = do

```
\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}\,\mathsf{T-Abs} inferTmLam :: ( AsSTLCType ty ty , AsSTLCTerm tm ty tm String
```

```
-> tm String
-> Maybe (m ty)
```

=> (tm String -> m ty)

inferTmLam infer tm = do

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

, MonadReader (M.Map String ty) m

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

-> Maybe (m ty)

inferTmLam infer tm = do

## $\frac{\Gamma, x_1 \colon T_1 \vdash t_2 \colon T_2}{\Gamma \vdash \lambda x_1 \colon T_1 \cdot t_2 \colon T_1 \to T_2} \mathsf{T-Abs}$

```
\Gamma \vdash \lambda x_1 \colon T_1. t_2 \colon T_1 \to T_2 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)

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

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

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

, Monad tm

=> (tm String -> m ty)

inferTmLam infer tm = do (n, ty1, s) <- preview \_TmLam

```
\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}
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 \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}
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 \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}
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 \colon T_1 \to T_2 \ \Gamma \vdash t_2 \colon T_1}{t_1 t_2 \colon T_2} \mathsf{T-App}
```

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

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

```
\frac{\Gamma \vdash t_1 \colon T_1 \to T_2 \ \Gamma \vdash t_2 \colon T_1}{t_1 t_2 \colon T_2} \mathsf{T-App}
```

inferTmApp infer tm = do

```
rac{\Gamma dash t_1 \colon T_1 	o T_2 \ \Gamma dash t_2 \colon T_1}{t_1 t_2 \colon T_2}T-App
```

(tm1, tm2) <- preview \_TmApp tm
return \$ do</pre>

```
\frac{\Gamma \vdash t_1 \colon T_1 \to T_2 \ \Gamma \vdash t_2 \colon T_1}{t_1 t_2 \colon 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
            \Rightarrow (tm a \rightarrow m ty) \rightarrow tm a \rightarrow Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
```

return \$ do

ty1 <- infer tm1 tv2 <- infer tm2

```
\frac{\Gamma \vdash t_1 \colon T_1 \to T_2 \ \Gamma \vdash t_2 \colon T_1}{t_1 t_2 \colon 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
            \Rightarrow (tm a \rightarrow m ty) \rightarrow tm a \rightarrow 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 \colon T_1 \to T_2 \ \Gamma \vdash t_2 \colon T_1}{t_1 t_2 \colon 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
              \Rightarrow (tm a \rightarrow m ty) \rightarrow tm a \rightarrow Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
     ty1 <- infer tm1
     tv2 <- infer tm2
     case preview _TyArr ty1 of
       Nothing -> throwing _NotArrow (ty1, ty2)
```

```
\frac{\Gamma \vdash t_1 \colon T_1 \to T_2 \ \Gamma \vdash t_2 \colon T_1}{t_1 t_2 \colon 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
              \Rightarrow (tm a \rightarrow m ty) \rightarrow tm a \rightarrow Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
     ty1 <- infer tm1
     tv2 <- infer tm2
     case preview _TyArr ty1 of
       Nothing -> throwing _NotArrow (ty1, ty2)
       Just (tyFrom, tyTo) -> do
```

```
\frac{\Gamma \vdash t_1 \colon T_1 \to T_2 \ \Gamma \vdash t_2 \colon T_1}{t_1 t_2 \colon 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
              \Rightarrow (tm a \rightarrow m ty) \rightarrow tm a \rightarrow Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
    ty1 <- infer tm1
    tv2 <- infer tm2
     case preview _TyArr ty1 of
       Nothing -> throwing _NotArrow (ty1, ty2)
       Just (tyFrom, tyTo) -> do
          expect tyFrom ty2
```

```
\frac{\Gamma \vdash t_1 \colon T_1 \to T_2 \ \Gamma \vdash t_2 \colon T_1}{t_1 t_2 \colon 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
              \Rightarrow (tm a \rightarrow m ty) \rightarrow tm a \rightarrow Maybe (m ty)
inferTmApp infer tm = do
  (tm1, tm2) <- preview _TmApp tm
  return $ do
    ty1 <- infer tm1
    tv2 <- 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 ...

