Tour of Hell

Haskell dialect scripting language in 1k lines

Why

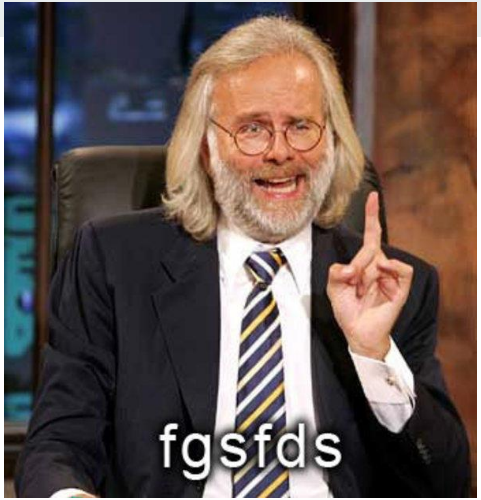
New Year’s Resolution

Write more shell scripts!

Bash downsides

Bash, zsh, fish, etc. have the same problems:

1. Incomprehensible gobbledegook.
2. They use quotation: x=$(ls -1)
   a. Leads to bugs too easily
3. Leaning too heavily on processes to do basic things
   a. Arithmetic, equality, ordering, etc. are completely unprincipled



Bash upsides

- Stable
- Simple
- Works the same on every machine
- ***Stable!***

Defining shell scripts

Anatomy of a Shell scripting language

- Very basic; glue code
- Interpreted – run immediately, no (visible) compilation steps
- No apparent module system
- No apparent package system
- No abstraction capabilities (classes, data types, polymorphic functions, etc.)

Package and module systems are generally not stable

This might be why bash is so reliable, and Node, Python, Haskell are not!

The Scripting Threshold

- When you reach for a module system or a package system, or abstraction capabilities.
- When you want more than what’s in the standard library.

… you probably want a general purpose programming language.

Solution

Why Haskell dialect?

- I know Haskell. It’s my go-to.
- It has a good story about equality, ordering, etc.
- It has a good runtime capable of trivially doing concurrency.
- Garbage collected, no funny business.
- Distinguishes bytes and text properly.
- Can be compiled to a static Linux x86 binary.
- Performs well.
- Types!

Decisions

- Use a faithful Haskell syntax parser (HSE).
  - It’s better.
- No imports/modules/packages.
  - That’s code reuse and leads to madness.
- No recursion (simpler to implement).
- Type-classes (Eq, Ord, Show, Monad).
  - Needed for e.g. List.lookup and familiar equality things.
- No polytypes.
  - That’s a kind of abstraction.
- Use all the same names for things (List.lookup, Monad.forM, Async.race, etc.)
  - Re-use intuitions.

Short version: it works

Example

```
main = do
  let x = "Hello!"
  Text.putStrLn (Function.id x)
  let lengths = List.map Text.length ["foo", "mu"]
  IO.mapM_ (\i -> Text.putStrLn (Int.show i)) lengths
```

Long version: Compiler pipeline

Parser

Use haskell-src-exts package.

```
data Exp l
```

Haskell expressions.

Constructors

`Var l (QName l)`	variable
`OverloadedLabel l String`	Overloaded label #foo
`IPVar l (IPName l)`	implicit parameter variable
`Con l (QName l)`	data constructor
`Lit l (Literal l)`	literal constant
`InfixApp l (Exp l) (QOp l) (Exp l)`	infix application
`App l (Exp l) (Exp l)`	ordinary application
`NegApp l (Exp l)`	negation expression −*exp*

But then what?

Desugaring…

Detour: Basic eval in Haskell

Total, well-typed eval in Haskell (HOAS)

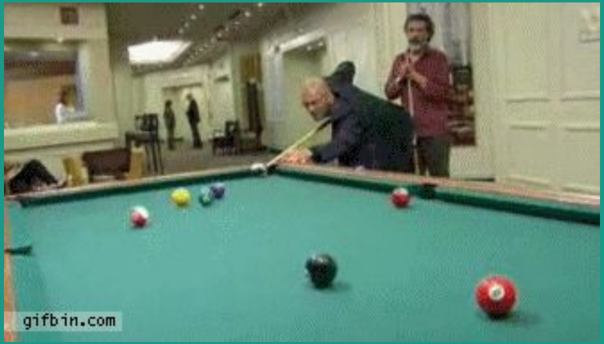
```haskell
-- λ> eval (A (L (\(C i) -> C (i * 2))) (C 2))
-- 4

{-# LANGUAGE GADTs #-}

data E a where
  C :: a -> E a
  L :: (E a -> E b) -> E (a -> b)
  A :: E (a -> b) -> E a -> E b

eval :: E a -> a
eval (L f) = \x -> eval (f (C x))
eval (A e1 e2) = (eval e1) (eval e2)
eval (C v) = v
```

This implementation is well-typed,

and doesn't crash.

Detour: Oleg Kiselyov's eval

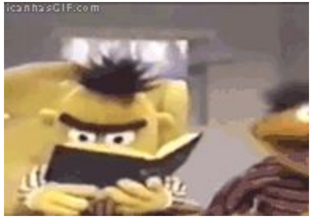
(From *Typed Tagless Final Interpreters*)



Type-indexed eval

```
data Var env t where
    VZ :: Var (t, env) t
    VS :: Var env t → Var (a, env) t
```

```
data Exp env t where
    B :: Bool             → Exp env Bool
    V :: Var env t        → Exp env t
    L :: Exp (a, env) b   → Exp env (a → b)
    A :: Exp env (a → b) → Exp env a → Exp env b
```




```
lookp :: Var env t → env → t
lookp VZ (x, _) = x
lookp (VS v) (_, env) = lookp v env
```

```
eval :: env → Exp env t → t
eval env (V v) = lookp v env
eval env (B b) = b
eval env (L e) = \x → eval (x, env) e
eval env (A e1 e2) = (eval env e1) (eval env e2)
```

Doesn't crash. The variables are statically indexed.

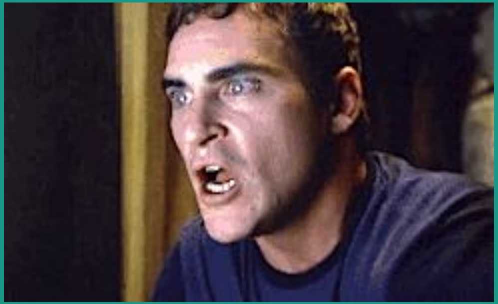
Hell's eval

```haskell
-- This is the entire evaluator. Type-safe and total.
eval :: env -> Term env t -> t
eval env (Var v) = lookp v env
eval env (Lam _ e) = \x -> eval (env, x) e
eval env (App e1 e2) = (eval env e1) (eval env e2)
eval _env (Lit a) = a

-- Type-safe, total lookup. The final @slot@ determines which slot of
-- a given tuple to pick out.
lookp :: Var env t -> env -> t
lookp (ZVar slot) (_, x) = slot x
lookp (SVar v) (env, x) = lookp v env
```

```haskell
data Term g t where
  Var :: Var g t -> Term g t
  Lam :: TypeRep (a :: Type) -> Term (g, a) b -> Term g (a -> b)
  App :: Term g (s -> t) -> Term g s -> Term g t
  Lit :: a -> Term g a

data Var g t where
  ZVar :: (t -> a) -> Var (h, t) a
  SVar :: Var h t -> Var (h, s) t
```

data Var env t **where**
  VZ :: Var (t, env) t
  VS :: Var env t → Var (a, env) t

Detour: Stephanie Weirich's type checker



```
tc :: UTerm -> exists ty. (Ty ty, Term ty)
```

A type checker with this type.

(Wrap it in an Either to avoid `error` calls, but minor detail.)

Untyped terms

```
data UTerm = UVar String
           | ULam String UType UTerm
           | UApp UTerm UTerm
           | UConBool Bool
           | UIf UTerm UTerm UTerm

data UType = UBool | UArr UType UType
```

Typed terms

```
data Term g t where
  Var :: Var g t -> Term g t
  Lam :: Ty a -> Term (g,a) b -> Term g (a->b)
  App :: Term g (s -> t) -> Term g s -> Term g t
  ConBool :: Bool -> Term g Bool
  If :: Term g Bool -> Term g a -> Term g a -> Term g a

data Var g t where
  ZVar :: Var (h,t) t
  SVar :: Var h t -> Var (h,s) t
```

Typecheck a type

```haskell
data Ty t where
  Bool :: Ty Bool
  Arr  :: Ty a -> Ty b -> Ty (a -> b)
```

```haskell
data ExType = forall t. ExType (Ty t)
```

```haskell
tcType :: UType -> ExType
tcType UBool = ExType Bool
tcType (UArr t1 t2) = case tcType t1 of { ExType t1' ->
                      case tcType t2 of { ExType t2' ->
                      ExType (Arr t1' t2') }}
```

```haskell
data UType = UBool | UArr UType UType
```

Typecheck an if

```haskell
tc :: UTerm -> TyEnv g -> Typed (Term g)

tc (UIf e1 e2 e3) env
  = case tc e1 env of { Typed Bool e1' ->
    case tc e2 env of { Typed t2   e2' ->
    case tc e3 env of { Typed t3   e3' ->
    case cmpTy t2 t3 of
        Nothing -> error "Type error"
        Just Equal -> Typed t2 (If e1' e2' e3') }}}
```

```haskell
data Equal a b where
  Equal :: Equal c c

cmpTy :: Ty a -> Ty b -> Maybe (Equal a b)
cmpTy Bool Bool = Just Equal
cmpTy (Arr a1 a2) (Arr b1 b2)
  = do  { Equal <- cmpTy a1 b1
        ; Equal <- cmpTy a2 b2
        ; return Equal }
```

```haskell
data Typed thing = forall ty. Typed (Ty ty) (thing ty)
```

(No error checking, imagine a _ -> error “Nooo!” branch)

Variables in scope

```
lookupVar :: String -> TyEnv g -> Typed (Var g)
lookupVar _ Nil = error "Variable not found"
lookupVar v (Cons s ty e)
  | v==s       = Typed ty ZVar
  | otherwise = case lookupVar v e of
                  Typed ty v -> Typed ty (SVar v)
```

```
-- The type environment and lookup
data TyEnv g where
  Nil  :: TyEnv g
  Cons :: String -> Ty t -> TyEnv h -> TyEnv (h,t)
```

```
tc (UVar v) env = case lookupVar v env of
                    Typed ty v -> Typed ty (Var v)
```

```
tc (ULam s ty body) env
  = case tcType ty of { ExType bndr_ty' ->
    case tc body (Cons s bndr_ty' env) of { Typed body_ty' body' ->
    Typed (Arr bndr_ty' body_ty')
          (Lam bndr_ty' body') }}
```

Applications, easy

```
tc (UApp e1 e2) env
  = case tc e1 env of { Typed (Arr bndr_ty body_ty) e1' ->
    case tc e2 env of { Typed arg_ty e2' ->
    case cmpTy arg_ty bndr_ty of
        Nothing -> error "Type error"
        Just Equal -> Typed body_ty (App e1' e2') }}
```

Type checker, review

```haskell
tc :: UTerm -> TyEnv g -> Typed (Term g)
tc (UVar v) env = case lookupVar v env of
                    Typed ty v -> Typed ty (Var v)
tc (UConBool b) env
  = Typed Bool (ConBool b)
tc (ULam s ty body) env
  = case tcType ty of { ExType bndr_ty' ->
    case tc body (Cons s bndr_ty' env) of { Typed body_ty' body' ->
    Typed (Arr bndr_ty' body_ty')
          (Lam bndr_ty' body') }}
tc (UApp e1 e2) env
  = case tc e1 env of { Typed (Arr bndr_ty body_ty) e1' ->
    case tc e2 env of { Typed arg_ty e2' ->
    case cmpTy arg_ty bndr_ty of
        Nothing -> error "Type error"
        Just Equal -> Typed body_ty (App e1' e2') }}
tc (UIf e1 e2 e3) env
  = case tc e1 env of { Typed Bool e1' ->
    case tc e2 env of { Typed t2   e2' ->
    case tc e3 env of { Typed t3   e3' ->
    case cmpTy t2 t3 of
        Nothing -> error "Type error"
        Just Equal -> Typed t2 (If e1' e2' e3') }}}
```

```haskell
showType :: Ty a -> String
showType Bool = "Bool"
showType (Arr t1 t2) = "(" ++ showType t1 ++ ") -> (" ++ showType t2 ++ ")"

uNot = ULam "x" UBool (UIf (UVar "x") (UConBool False) (UConBool True))

test :: UTerm
test = UApp uNot (UConBool True)

main = putStrLn (case tc test Nil of
                        Typed ty _ -> showType ty
               )
```

Evaluating Term

Easy – use Oleg’s type-indexed eval.

Detour: Eitan Chatav's type-class support

Preamble

```haskell
data U_Expr
  = U_Bool Bool
  | U_Int Int
  | U_Double Double
  | U_And U_Expr U_Expr
  | U_Add U_Expr U_Expr

data T_Expr x where
  T_Bool :: Bool -> T_Expr Bool
  T_Int :: Int -> T_Expr Int
  T_Double :: Double -> T_Expr Double
  T_And :: T_Expr Bool -> T_Expr Bool -> T_Expr Bool
  T_Add :: Num x => T_Expr x -> T_Expr x -> T_Expr x
deriving instance Show (T_Expr x)

data Type x where
  TypeBool :: Type Bool
  TypeInt :: Type Int
  TypeDouble :: Type Double

data (:::) f g = forall x. Typeable x => (:::) (f x) (g x)
```

```haskell
data Typed thing = forall ty. Typed (Ty ty) (thing ty)
```

Type-class instance resolving

```
check :: U_Expr -> Maybe (T_Expr ::: Type)
check expr = case expr of
  U_Bool x -> return $ T_Bool x ::: TypeBool
  U_Int x -> return $ T_Int x ::: TypeInt
  U_Double x -> return $ T_Double x ::: TypeDouble
  U_And x y -> do
    tx ::: tyx <- check x
    ty ::: tyy <- check y
    HRefl <- eqTypeRep (typeOf tyx) (typeOf tyy)
    HRefl <- eqTypeRep (typeOf tyx) (typeOf TypeBool)
    return $ T_And tx ty ::: TypeBool
  U_Add x y -> do
    tx ::: tyx <- check x
    ty ::: tyy <- check y
    HRefl <- eqTypeRep (typeOf tyx) (typeOf tyy)
    Dict <- checkNum tyx
    return $ T_Add tx ty ::: tyx

    where checkNum :: Type x -> Maybe (Dict (Num x))
          checkNum TypeInt = Just Dict
          checkNum TypeDouble = Just Dict
          checkNum _ = Nothing
```

Type.Reflection

Reminder: typed AST

```
data Term g t where
  Var :: Var g t -> Term g t
  Lam :: TypeRep (a :: Type) -> Term (g, a) b -> Term g (a -> b)
  App :: Term g (s -> t) -> Term g s -> Term g t
  Lit :: a -> Term g a

data Var g t where
  ZVar :: (t -> a) -> Var (h, t) a
  SVar :: Var h t -> Var (h, s) t
```

Type.Reflection

```
eqTypeRep :: forall k1 k2 (a :: k1) (b :: k2). TypeRep a -> TypeRep b -> Maybe (a :~~: b)
```

```
typeOf :: Typeable a => a -> TypeRep a
```

```
data TypeRep (a :: k)
```
Source

TypeRep is a concrete representation of a (monomorphic) type. TypeRep supports reasonably efficient equality. See Note [Grand plan for Typeable] in GHC.Tc.Instance.Typeable

▽ Instances

- ▹ `TestEquality (TypeRep :: k -> Type)` # Source
- ▹ `Show (TypeRep a)` # Source
- ▹ `Eq (TypeRep a)` # Source — *Since: base-2.1*
- ▹ `Ord (TypeRep a)` # Source — *Since: base-4.4.0.0*

```
data SomeTypeRep where
```

A non-indexed type representation.

Constructors

```
SomeTypeRep :: forall k (a :: k). !(TypeRep a) -> SomeTypeRep
```

```
typeRepKind :: forall k (a :: k). TypeRep a -> TypeRep k
```

Type application

```haskell
-- | Supports up to 3-ary type functions, but not more.
applyTypes :: SomeTypeRep -> SomeTypeRep -> Maybe SomeTypeRep
applyTypes (SomeTypeRep f) (SomeTypeRep a) = do
  Type.HRefl <- Type.eqTypeRep (typeRepKind a) (typeRep @Type)
  if
    | Just Type.HRefl <- Type.eqTypeRep (typeRepKind f) (typeRep @(Type -> Type)) ->
      pure $ SomeTypeRep $ Type.App f a
    | Just Type.HRefl <- Type.eqTypeRep (typeRepKind f) (typeRep @(Type -> Type -> Type)) ->
      pure $ SomeTypeRep $ Type.App f a
    | Just Type.HRefl <- Type.eqTypeRep (typeRepKind f) (typeRep @(Type -> Type -> Type -> Type)) ->
      pure $ SomeTypeRep $ Type.App f a
    | Just Type.HRefl <- Type.eqTypeRep (typeRepKind f) (typeRep @(Type -> Type -> Type -> Type -> Type)) ->
      pure $ SomeTypeRep $ Type.App f a
    | otherwise -> Nothing
```

Hell’s untyped AST

Desugarer type

```
desugarExp :: Map String (UTerm ()) -> HSE.Exp HSE.SrcSpanInfo ->
   Either DesugarError (UTerm ())
desugarExp globals = go where
  go = \case
    HSE.Paren _ x -> go x
    HSE.If _ i t e ->
      (\e' t' i' -> UApp () (UApp () (UApp () bool' e') t') i')
        <$> go e <*> go t <*> go i
```

New type checker signature

For type inference

```
data UTerm t
  = UVar t String
  | ULam t Binding (Maybe SomeStarType) (UTerm t)
  | UApp t (UTerm t) (UTerm t)
```

(missing constructor here)

```
data SomeStarType = forall (a :: Type). SomeStarType (TypeRep a)
```

```
-- Type check a term given an environment of names.
tc :: (UTerm SomeTypeRep) -> TyEnv g -> Typed (Term g)
tc (UVar _ v) env = case lookupVar v env of
  Typed ty v -> Typed ty (Var v)
```

But otherwise basically the same.

```
toStarType :: SomeTypeRep -> Maybe SomeStarType
toStarType (SomeTypeRep t) = do
  Type.HRefl <- Type.eqTypeRep (typeRepKind t) (typeRep @Type)
  pure $ SomeStarType t
```

Type inference

Inference type

```haskell
data IRep v
  = IVar v
  | IApp (IRep v) (IRep v)
  | IFun (IRep v) (IRep v)
  | ICon SomeTypeRep
  deriving (Functor, Traversable, Foldable, Eq, Ord, Show)
```

Top-level: normal stuff

```haskell
-- | Note: All types in the input are free of metavars. There is an
-- intermediate phase in which there are metavars, but then they're
-- all eliminated. By the type system, the output contains only
-- determinate types.
inferExp ::
  Map String (UTerm SomeTypeRep) ->
  UTerm () ->
  Either InferError (UTerm SomeTypeRep)
inferExp _ uterm =
  case unify equalities of
    Left unifyError -> Left $ UnifyError unifyError
    Right subs ->
      case traverse (zonkToStarType subs) iterm of
        Left zonkError -> Left $ ZonkError $ zonkError
        Right sterm -> pure sterm

  where (iterm, equalities) = elaborate uterm
```

```haskell
-- | Zonk a type and then convert it to a type: t :: *
zonkToStarType :: Map IMetaVar (IRep IMetaVar) -> IRep IMetaVar -> Either ZonkError SomeTypeRep
zonkToStarType subs irep = do
  zonked <- zonk (substitute subs irep)
  toSomeTypeRep zonked
```

```haskell
-- | Remove any metavars from the type.
--
-- <https://stackoverflow.com/questions/31889048/what-does-the-ghc-source-mean-by-zonk>
zonk :: IRep IMetaVar -> Either ZonkError (IRep Void)
zonk = \case
  IVar v -> Left AmbiguousMetavar
  ICon c -> pure $ ICon c
  IFun a b -> IFun <$> zonk a <*> zonk b
  IApp a b -> IApp <$> zonk a <*> zonk b
```

```haskell
-- | A complete implementation of conversion from the inferer's type
-- rep to some star type, ready for the type checker.
toSomeTypeRep :: IRep Void -> Either ZonkError SomeTypeRep
```

Elaboration

Pretty normal stuff here, too.

```haskell
data Equality a = Equality a a
  deriving (Show, Functor)
```

```haskell
equal :: MonadState Elaborate m => IRep IMetaVar -> IRep IMetaVar -> m ()
equal x y = modify \elaborate -> elaborate { equalities = equalities elaborate <> Set.singleton (Equality x y) }

freshIMetaVar :: MonadState Elaborate m => m IMetaVar
freshIMetaVar = do
  Elaborate{counter} <- get
  modify \elaborate -> elaborate { counter = counter + 1 }
  pure $ IMetaVar0 counter
```

```haskell
-- | Elaboration phase.
--
-- Note: The input term contains no metavars. There are just some
-- UForalls, which have poly types, and those are instantiated into
-- metavars.
--
-- Output type /does/ contain meta vars.
elaborate :: UTerm () -> (UTerm (IRep IMetaVar), Set (Equality (IRep IMetaVar)))
```

Easy ones

```haskell
-- | Convert from a type-indexed type to an untyped type.
fromSomeStarType :: forall void. SomeStarType -> IRep void
fromSomeStarType (SomeStarType typeRep) = go typeRep where
  go :: forall a. TypeRep a -> IRep void
  go = \case
    Type.Fun a b -> IFun (go a) (go b)
    Type.App a b -> IApp (go a) (go b)
    typeRep@Type.Con{} -> ICon (SomeTypeRep typeRep)
```

```haskell
go = \case
  UVar () string -> do
    env <- ask
    ty <- case Map.lookup string env of
            Just typ -> pure typ
            Nothing -> fmap IVar freshIMetaVar
    pure $ UVar ty string
  UApp () f x -> do
    f' <- go f
    x' <- go x
    b <- fmap IVar freshIMetaVar
    equal (typeOf f') (IFun (typeOf x') b)
    pure $ UApp b f' x'
  ULam () binding mstarType body -> do
    a <- case mstarType of
      Just ty -> pure $ fromSomeStarType ty
      Nothing -> fmap IVar freshIMetaVar
    vars <- bindingVars a binding
    body' <- local (Map.union vars) $ go body
    let ty = IFun a (typeOf body')
    pure $ ULam ty binding mstarType body'
```

Unification

Normal stuff, nothing interesting here at all.

Same as typing haskell in haskell.

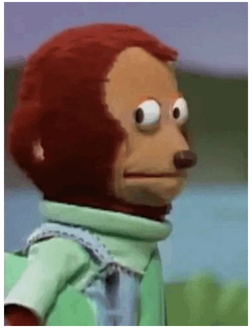
```
-- | Unification of equality constraints, a ~ b, to substitutions.
unify :: Set (Equality (IRep IMetaVar)) -> Either UnifyError (Map IMetaVar (IRep IMetaVar))
```

```
-- | Unification of equality constraints, a ~ b, to substitutions.
unify :: Set (Equality (IRep IMetaVar)) -> Either UnifyError (Map IMetaVar (IRep IMetaVar))
unify = foldM update mempty where
  update existing equality =
    fmap (`extends` existing)
         (examine (fmap (substitute existing) equality))
  examine (Equality a b)
   | a == b = pure mempty
   | IVar ivar <- a = bindMetaVar ivar b
   | IVar ivar <- b = bindMetaVar ivar a
   | IFun a1 b1 <- a,
     IFun a2 b2 <- b =
       unify (Set.fromList [Equality a1 a2, Equality b1 b2])
   | IApp a1 b1 <- a,
     IApp a2 b2 <- b =
       unify (Set.fromList [Equality a1 a2, Equality b1 b2])
   | ICon x <- a, ICon y <- b =
      if x == y then pure mempty
                else Left $ TypeConMismatch x y
   | otherwise = Left $ TypeMismatch a b

-- | Apply new substitutions to the old ones, and expand the set to old+new.
extends :: Map IMetaVar (IRep IMetaVar) -> Map IMetaVar (IRep IMetaVar) -> Map IMetaVar (IRep IMetaVar)
extends new old = fmap (substitute new) old <> new

-- | Apply any substitutions to the type, where there are metavars.
substitute :: Map IMetaVar (IRep IMetaVar) -> IRep IMetaVar -> IRep IMetaVar
substitute subs = go where
  go = \case
    IVar v -> case Map.lookup v subs of
      Nothing -> IVar v
      Just ty -> ty
    ICon c -> ICon c
    IFun a b -> IFun (go a) (go b)
    IApp a b -> IApp (go a) (go b)

-- | Do an occurrs check, if all good, return a binding.
bindMetaVar :: IMetaVar -> IRep IMetaVar
            -> Either UnifyError (Map IMetaVar (IRep IMetaVar))
bindMetaVar var typ
  | occurs var typ = Left OccursCheck
  | otherwise = pure $ Map.singleton var typ

-- | Occurs check.
occurs :: IMetaVar -> IRep IMetaVar -> Bool
occurs ivar = any (==ivar)
```

Polymorphic primitives

Forall

```haskell
data UTerm t
  = UVar t String
  | ULam t Binding (Maybe SomeStarType) (UTerm t)
  | UApp t (UTerm t) (UTerm t)

  -- IRep below: The variables are poly types, they aren't metavars,
  -- and need to be instantiated.
  | UForall t [SomeStarType] Forall [TH.Uniq] (IRep TH.Uniq) [t]
  deriving (Traversable, Functor, Foldable)
```



```haskell
data Forall where
  NoClass :: (forall (a :: Type). TypeRep a -> Forall) -> Forall
  OrdEqShow :: (forall (a :: Type). (Ord a, Eq a, Show a) => TypeRep a -> Forall) -> Forall
  Monadic :: (forall (m :: Type -> Type). (Monad m) => TypeRep m -> Forall) -> Forall
  Final :: (forall g. Typed (Term g)) -> Forall

lit :: Type.Typeable a => a -> UTerm ()
lit l = UForall () [] (Final (Typed (Type.typeOf l) (Lit l))) [] (fromSomeStarType (SomeStarType (Type.typeOf l))) []
```

Example

```
id = NoClass (\(TypeRep :: TypeRep a) -> Final (lit (id :: a -> a)))
```

Type-checking Foralls

Yes this actually works.

```
tc (UForall _ _ fall _ _ reps) _env = go reps fall where
  go :: [SomeTypeRep] -> Forall -> Typed (Term g)
  go [] (Final typed) = typed
  go (StarTypeRep rep:reps) (NoClass f) = go reps (f rep)
  go (StarTypeRep rep:reps) (OrdEqShow f) =
    if
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @Int) -> go reps (f rep)
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @Bool) -> go reps (f rep)
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @Char) -> go reps (f rep)
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @Text) -> go reps (f rep)
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @ByteString) -> go reps (f rep)
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @ExitCode) -> go reps (f rep)
        | otherwise -> error $ "type doesn't have enough instances " ++ show rep
  go (SomeTypeRep rep:reps) (Monadic f) =
    if
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @IO) -> go reps (f rep)
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @Maybe) -> go reps (f rep)
        | Just Type.HRefl <- Type.eqTypeRep rep (typeRep @[]) -> go reps (f rep)
        | Type.App either a <- rep,
          Just Type.HRefl <- Type.eqTypeRep either (typeRep @Either) -> go reps (f rep)
        | otherwise -> error $ "type doesn't have enough instances " ++ show rep
  go _ _ = error "forall type arguments mismatch."
```

Inferring foralls

```
UForall () types forall' uniqs polyRep _ -> do
  -- Generate variables for each unique.
  vars <- for uniqs \uniq -> do
    v <- freshIMetaVar
    pure (uniq, v)
  -- Fill in the polyRep with the metavars.
  monoType <- for polyRep \uniq ->
    case List.lookup uniq vars of
      Nothing -> error "Instantiation is broken internally."
      Just var -> pure var
  -- Order of types is position-dependent, apply the ones we have.
  for (zip vars types) \((_uniq, var), someTypeRep) ->
    equal (fromSomeStarType someTypeRep) (IVar var)
  -- Done!
  pure $ UForall monoType types forall' uniqs polyRep (map (IVar . snd) vars)
```

Primops

```haskell
("Text.isInfixOf", lit Text.isInfixOf),
-- Int operations
("Int.show", lit (Text.pack . show @Int)),
("Int.eq", lit ((==) @Int)),
("Int.plus", lit ((+) @Int)),
("Int.subtract", lit (subtract @Int)),
-- Bytes I/O
("ByteString.hGet", lit ByteString.hGet),
("ByteString.hPutStr", lit ByteString.hPutStr),
("ByteString.readProcess", lit b_readProcess),
("ByteString.readProcess_", lit b_readProcess_),
("ByteString.readProcessStdout_", lit b_readProcessStdout_),
-- Handles, buffering
("IO.stdout", lit IO.stdout),
("IO.stderr", lit IO.stderr),
("IO.stdin", lit IO.stdin),
("IO.hSetBuffering", lit IO.hSetBuffering),
("IO.NoBuffering", lit IO.NoBuffering),
("IO.LineBuffering", lit IO.LineBuffering),
("IO.BlockBuffering", lit IO.BlockBuffering),
-- Bool
("Bool.True", lit Bool.True),
("Bool.False", lit Bool.False),
("Bool.not", lit Bool.not),
-- Get arguments
("Environment.getArgs", lit $ fmap (map Text.pack) getArgs),
("Environment.getEnvironment", lit $ fmap (map (bimap Text.pack Text.pack)) getEnvironment),
("Environment.getEnv", lit $ fmap Text.pack . getEnv . Text.unpack),
-- Current directory
("Directory.createDirectoryIfMissing", lit (\b f -> Dir.createDirectoryIfMissing b (Text.unpac
("Directory.createDirectory", lit (Dir.createDirectory . Text.unpack)),
("Directory.getCurrentDirectory", lit (fmap Text.pack Dir.getCurrentDirectory)),
("Directory.listDirectory", lit (fmap (fmap Text.pack) . Dir.listDirectory . Text.unpack)),
("Directory.setCurrentDirectory", lit (Dir.setCurrentDirectory . Text.unpack)),
("Directory.renameFile", lit (\x y -> Dir.renameFile (Text.unpack x) (Text.unpack y))),
("Directory.copyFile", lit (\x y -> Dir.copyFile (Text.unpack x) (Text.unpack y))),
```

Poly: Template Haskell

```
id =
  NoClass (\(TypeRep :: TypeRep a) ->
    Final (lit (id :: a -> a)))
```

```
-- Monad
"Monad.bind" (Prelude.>>=) :: forall m a b. Monad m => m a -> (a -> m b) -> m b
"Monad.then" (Prelude.>>) :: forall m a b. Monad m => m a -> m b -> m b
"Monad.return" return :: forall a m. Monad m => a -> m a
-- Monadic operations
"Monad.mapM_" mapM_ :: forall a m. Monad m => (a -> m ()) -> [a] -> m ()
"Monad.forM_" forM_ :: forall a m. Monad m => [a] -> (a -> m ()) -> m ()
"Monad.mapM" mapM :: forall a b m. Monad m => (a -> m b) -> [a] -> m [b]
"Monad.forM" forM :: forall a b m. Monad m => [a] -> (a -> m b) -> m [b]
"Monad.when" when :: forall m. Monad m => Bool -> m () -> m ()
-- IO
"IO.mapM_" mapM_ :: forall a. (a -> IO ()) -> [a] -> IO ()
"IO.forM_" forM_ :: forall a. [a] -> (a -> IO ()) -> IO ()
"IO.pure" pure :: forall a. a -> IO a
"IO.print" (t_putStrLn . Text.pack . Show.show) :: forall a. Show a => a -> IO ()
-- Show
"Show.show" (Text.pack . Show.show) :: forall a. Show a => a -> Text
-- Eq/Ord
"Eq.eq" (Eq.==) :: forall a. Eq a => a -> a -> Bool
"Ord.lt" (Ord.<) :: forall a. Ord a => a -> a -> Bool
"Ord.gt" (Ord.>) :: forall a. Ord a => a -> a -> Bool
-- Tuples
"Tuple.(,)" (,) :: forall a b. a -> b -> (a,b)
"Tuple.(,)" (,) :: forall a b. a -> b -> (a,b)
"Tuple.(,,)" (,,) :: forall a b c. a -> b -> c -> (a,b,c)
"Tuple.(,,,)" (,,,) :: forall a b c d. a -> b -> c -> d -> (a,b,c,d)
-- Exceptions
"Error.error" (error . Text.unpack) :: forall a. Text -> a
-- Bool
"Bool.bool" Bool.bool :: forall a. a -> a -> Bool -> a
-- Function
"Function.id" Function.id :: forall a. a -> a
"Function.fix" Function.fix :: forall a. (a -> a) -> a
-- Lists
"List.cons" (:) :: forall a. a -> [a] -> [a]
"List.nil" [] :: forall a. [a]
```

Supported types

No need to explicitly mention all

The details of the types.

```haskell
supportedTypeConstructors :: Map String SomeTypeRep
supportedTypeConstructors = Map.fromList [
  ("Bool", SomeTypeRep $ typeRep @Bool),
  ("Int", SomeTypeRep $ typeRep @Int),
  ("Char", SomeTypeRep $ typeRep @Char),
  ("Text", SomeTypeRep $ typeRep @Text),
  ("ByteString", SomeTypeRep $ typeRep @ByteString),
  ("ExitCode", SomeTypeRep $ typeRep @ExitCode),
  ("Maybe", SomeTypeRep $ typeRep @Maybe),
  ("Either", SomeTypeRep $ typeRep @Either),
  ("IO", SomeTypeRep $ typeRep @IO),
  ("ProcessConfig", SomeTypeRep $ typeRep @ProcessConfig)
  ]
```

Desugarer points

do-notation

```
HSE.Do _ stmts -> do
  let loop f [HSE.Qualifier _ e] = f <$> go e
      loop f (s:ss) = do
        case s of
          HSE.Generator _ pat e -> do
             (s, rep) <- desugarArg pat
             m <- go e
             loop (f . (\f -> UApp () (UApp () bind' m) (ULam () s rep f))) ss
          HSE.LetStmt _ (HSE.BDecls _ [HSE.PatBind _ pat (HSE.UnGuardedRhs _ e) Nothing]) -> do
             (s, rep) <- desugarArg pat
             value <- go e
             loop (f . (\f -> UApp () (ULam () s rep f) value)) ss
          HSE.Qualifier _ e -> do
            e' <- go e
            loop (f . UApp () (UApp () then' e')) ss
      loop _ _ = error "Malformed do-notation!"
  loop id stmts
```

Frontend

Main runner

```
dispatch :: Command -> IO ()
dispatch Version = putStrLn "2023-12-12"
dispatch (Run filePath) = do
  string <- readFile filePath
  case HSE.parseModuleWithMode HSE.defaultParseMode { HSE.extensions = HSE.extensions HSE.defaultParseMode ++ [HSE.Er
    HSE.ParseFailed _ e -> error $ e
    HSE.ParseOk binds
      | anyCycles binds -> error "Cyclic bindings are not supported!"
      | otherwise ->
            case desugarAll binds of
              Left err -> error $ "Error desugaring! " ++ show err
              Right terms ->
                case lookup "main" terms of
                  Nothing -> error "No main declaration!"
                  Just main' ->
                    case inferExp mempty main' of
                      Left err -> error $ "Error inferring! " ++ show err
                      Right uterm ->
                        case check uterm Nil of
                           Typed t ex ->
                             case Type.eqTypeRep (typeRepKind t) (typeRep @Type) of
                               Just Type.HRefl ->
                                 case Type.eqTypeRep t (typeRep @(IO ())) of
                                   Just Type.HRefl ->
                                     let action :: IO () = eval () ex
                                     in action
                                   Nothing -> error $ "Type isn't IO (), but: " ++ show t
```

FIN