Overview

1. Features
2. Abstractions used.
3. Type inference for statements (returns).
4. What is not working.

Features

- Type inference
- Let polymorphism
- Let bindings allow for recursion
  - But no mutual recursion. Things can only refer to themselves and previously defined things.
- Functions are required by type system to return a value.
  - Returns automatically inserted when "missing".

Substitution and Inference in Haskell

We have to do a lot of work with substitutions and inference for this part of the compiler, so we come up with abstractions for easing this.

- A typeclass for things we can perform substitution on
- A monad for doing inference

Inference Monad

During type inference we need to keep track of

- A context
- The next fresh variable
- Information about returns (ignore this for now)

For this we use the following monad transformer stack:

```
data InfState = InfState {
    ctx :: [TypeContext],
    freshNum :: Int ,
    rets :: [Bool]
    } deriving (Show)

data TypeError = NoUnify Type Type | NotInContext Identifier deriving (Sho

type Inference a = StateT InfState (Either TypeError) a
```

Inference Monad

This allows us to write code like:

```
typeInferExp (Op2E o e1 e2) ctx t = do
    s1 <- typeInferExp e1 ctx (opInType o)
    s2 <- typeInferExp e2 (subst s1 ctx) (opInType o)
    s3 <- lift $ mgu (subst (s1 <> s2) t) (opOutType o)
    pure (s1 <> s2 <> s3)
```

Note: This is from an older version where contexts where not part of the state, but passed explicitly.

Inference Monad

We also have functions for dealing with the context (stack)

```
ctxLookup :: String -> Inference TypeScheme
ctxAdd :: (String, TypeScheme) -> Inference ()
pushCtx :: Inference ()
popCtx :: Inference ()
getCtx :: Inference TypeContext
```

Substitutions

Looking at the unification and type inference algorithms, we observe that we have to perform substitutions on many things. e.g:

$$* = \mathcal{M}(\Gamma^{*_1}, e_2, \sigma^{*_1}) \circ *_1$$

Writing this in a way more like the following would be cool (maybe?):

$$* = \mathcal{M}^{*_1}(\Gamma, e_2, \sigma)$$

```
class Substable a where
    subst :: Substitution -> a -> a
```

Substitutions

Pretty much everything can be substituted. Uninteresting implementations omitted.

```
instance Substable Type where ...
instance Substable TypeContext where ...
instance Substable TypeScheme where ...

instance Substable Substitution where
    subst s1 s2 = s1 <> s2
instance (Substable a, Substable b) => Substable (a -> b) where
    subst s f = \a -> subst s (f (subst s a))

instance Substable InfState where ...
instance (Substable a) => Substable (Either e a) where ...
instance (Substable a, Substable s, Functor m)
        => Substable (StateT s m a) where
    subst s = fmap (subst s) . withStateT (subst s)
```

Substitutions

Now we can write our example like this:

```
typeInferExp (Op2E o e1 e2) t = do
      s1 <- typeInferExp e1 (opInType o)
      s2 <- (subst s1 (typeInferExp e2)) (opInType o)
       lift $ (subst s2 mgu) t (opOutType o)
```

With the following definiton

```
typeInferExp' e t s = subst s (typeInferExp e) t
```

We can even do this:

```
typeInferExp (Op2E o e1 e2) t =
      typeInferExp  e1 (opInType o) >>=
      typeInferExp' e2 (opInType o) >>=
       lift  . (mgu' t (opOutType o))
```

Statements

Example:

```
typeInferStmtList ((AssignS id fs e):ss) t = do
    schm@(TypeScheme bounds _) <- ctxLookup id
    vs <- replicateM (length bounds) freshVar
    s <- typeInferExp e (concrete schm vs)
    typeInferStmtList' ss t s
```

Returns

A challenge that is unique to statements, and is not easy to express in terms of type inference on a λ-calculus like language is return statements.

1. All returns must be of the same type.
2. Every function must return something.

1 is easy. Just let the type of return statements be the type of the expressions the return, and ignore the type of other statements.
2 requires something extra. If we don't make sure of this:

- Functions with no annotations and no returns will have return type $\forall a.a$ due to the way we handle 1. This is a runtime error.
- Will this be a problem for functions with return type restricted by type annotations?

Making sure functions return

Solution: Keep a stack of booleans in the state.

- Initially the stack is [False].
- When a return is encountered set the top element to True.
- When entering a branch of a control structure push False on the stack.
- When leaving *the last* branch of a control structure pop the number of branches elements from the stack. Then:
  - let x be the conjunction of the popped elements.
  - If one of the branches is guarenteed to be executed set the top element to be the disjunction of itself and x.

Returns

The Inference

```
implicitReturnNeeded :: Inference Bool
```

returns(infers?) true only if the stack is exactly [False]. This means we are at the top level of a function and a return is not guarenteed.

Returns – Example 1

```
typeInferStmtList (IfS e body1 (Just body2):ss) t = do
     s1 <- typeInferExp e TBool
     enterControl
     pushCtx
     s2 <- typeInferStmtList' body1 t s1
     popCtx
     enterControl
     pushCtx
     s3 <- typeInferStmtList' body2 t s2
     popCtx
     leaveControl 2 True
     typeInferStmtList' ss t s3
```

Returns – Example 2

```
typeInferStmtList [] t = do
    b <- implicitReturnNeeded
    if b then
        lift $ mgu t TVoid
    else
        pure mempty
```

Things which are missing

- Type annotations