CSE130 - WI19

PA5 Discussion

Agenda

- Tips on *Warm-Up*
- Tips on *Unification*
- Tips on *Inference*

Warm-Up ToDo

```
freeTVars    :: a -> [TVar]

lookupTVar   :: TVar -> Subst -> Type

removeTVar   :: TVar -> Subst -> Subst

apply        :: Subst -> a -> a

extendSubst :: Subst -> TVar -> Type -> Subst
```

Warm-Up ToDo

```
freeTVars    :: a -> [TVar]

lookupTVar   :: TVar -> Subst -> Type

removeTVar   :: TVar -> Subst -> Subst

apply        :: Subst -> a -> a

extendSubst :: Subst -> TVar -> Type -> Subst
```

freeTVars

```
-- | Type variables of a type
instance HasTVars Type where
  freeTVars t     = error "TBD: type freeTVars"

-- | Free type variables of a poly-type (remove forall-bound vars)
instance HasTVars Poly where
  freeTVars s     = error "TBD: poly freeTVars"
```

`freeTVars :: Type -> [TVar]`

How to implement

1. Pattern-match the `Type` constructors
2. **NOT** all `Type` constructors have free type variables. Which of them do not? `TInt` is one of them
   a. Return the `[]` for these cases
3. The trickiest case is handling the `| Type1 :=> Type2` constructor.
   a. Here you'll have two inner constructors to handle
   b. Handle duplicates!

```
data Type
  = TInt                -- Int
  | TBool               -- Bool
  | Type :=> Type       -- T1 -> T2
  | TVar TVar           -- a, b, c
  | TList Type          -- [T]
  deriving (Eq, Ord)
```

freeTVars :: Poly -> [TVar]

How to implement

1. Pattern-match the `Poly` constructors
2. Call freeTVars recursively
3. One of these `Poly` constructors **has bound variables.** Which one is it? A bounded variable is not free (definition) so make sure to remove them!

```
data Poly = Mono Type
          | Forall TVar Poly
```

Warm-Up ToDo

```
freeTVars   :: a -> [TVar]

lookupTVar  :: TVar -> Subst -> Type

removeTVar  :: TVar -> Subst -> Subst

apply       :: Subst -> a -> a

extendSubst :: Subst -> TVar -> Type -> Subst
```

```
lookupTVar  :: TVar -> Subst -> Type

removeTVar  :: TVar -> Subst -> Subst
```

How to implement

1. The `Subst` parameter is just a list. You know how to traverse these in Haskell. **Hint:** use recursion!

2. The main **trick** is that, in `removeTVar` you're building a list that is (potentially) skipping an element from the original list.

Warm-Up ToDo

```
freeTVars    :: a -> [TVar]

lookupTVar   :: TVar -> Subst -> Type

removeTVar   :: TVar -> Subst -> Subst

apply        :: Subst -> a -> a

extendSubst :: Subst -> TVar -> Type -> Subst
```

```
apply :: Subst -> a -> a
```

How to implement

1. Pattern-match all constructors in `Type` and `Poly`

2. You will have to re-use `lookupTVar` and `removeTVar` but not necessarily both of them for the same data class (`Type` and `Poly`)

3. Structurally similar to the implementation of `freeTVars`

Warm-Up ToDo

```
freeTVars    :: a -> [TVar]

lookupTVar   :: TVar -> Subst -> Type

removeTVar   :: TVar -> Subst -> Subst

apply        :: Subst -> a -> a

extendSubst :: Subst -> TVar -> Type -> Subst
```

```
extendSubst :: Subst -> TVar -> Type -> Subst
```

How to implement

1. Can be a one-liner

2. Re-use the `apply` to propagate the newly added substitution information to pre-existing tuples in the array

Agenda

- Tips on *Warm-Up*
- Tips on *Unification*
- Tips on *Inference*

Unification ToDo

```
unifyTVar :: InferState -> TVar -> Type -> InferState

unify :: InferState -> Type -> Type -> InferState
```

Unification ToDo

```
unifyTVar :: InferState -> TVar -> Type -> InferState

unify :: InferState -> Type -> Type -> InferState
```

```
unifyTVar :: InferState -> TVar -> Type -> InferState
```

How to implement

1. Super simple
2. 3 cases
   a. Unify “a” with “a” <= In README
   b. Unify “a” with a type containing a free-var “a” <= In README
   c. Unify “a” with a type not containing a free-var “a” <= you’ll use extendState

Unification ToDo

```
unifyTVar :: InferState -> TVar -> Type -> InferState
```

`unify :: InferState -> Type -> Type -> InferState`

```
unify :: InferState -> Type -> Type -> InferState
```

How to implement the trickiest parts

1. When either `Type` argument is a `TVar`, then delegate to `unifyTVar`
2. The *trickiest* case is when both `Type` arguments are `Type1 :=> Type2`.
   a. Unify both `Type1`s.
   b. Propagate the newfound substitutions onto the `Type2`. You should already know what method does this
   c. Unify both `Type2`s.

Agenda

- Tips on *Warm-Up*
- Tips on *Unification*
- Tips on *Inference*

Type Inference ToDo

```
generalize :: TypeEnv -> Type -> Poly

instantiate :: Int -> Poly -> (Int, Type)

infer :: InferState -> TypeEnv -> Expr -> (InferState, Type)
```

Type Inference ToDo

```
generalize :: TypeEnv -> Type -> Poly

instantiate :: Int -> Poly -> (Int, Type)

infer :: InferState -> TypeEnv -> Expr -> (InferState, Type)
```

```
generalize :: TypeEnv -> Type -> Poly
```

How to implement

1. Get all free type variables from the type that do not appear in the enviroment. Use `freeTVars` to get this

2. Make sure to remove duplicate free variables

3. Add ForAlls for all these type variables. Recursion and/or folding are your friends.

Type Inference ToDo

```
generalize :: TypeEnv -> Type -> Poly
```

`instantiate :: Int -> Poly -> (Int, Type)`

```
infer :: InferState -> TypeEnv -> Expr -> (InferState, Type)
```

`instantiate :: Int -> Poly -> (Int, Type)`

How to implement

1. You may need a helper function to keep track of fresh variables.
2. 2 cases: Mono and Poly
3. Poly case: add new fresh variable for the bounded type variable to the enviroment (`freshTV`) is your friend. Don’t forget to increase the counter
4. Mono case: propagation substitutions w/ `apply`

Type Inference ToDo

```
generalize :: TypeEnv -> Type -> Poly

instantiate :: Int -> Poly -> (Int, Type)
```

`infer :: InferState -> TypeEnv -> Expr -> (InferState, Type)`

```
infer :: InferState -> TypeEnv -> Expr -> (InferState, Type)
```

General Strategy

1. I can't give much away here
2. The lecture notes help *a lot*
3. **Generalize** in the let case
4. **Extend** the type enviroment in Let and Lam cases
5. In EBin and EIf, construct expressions that use your Prelude types
6. Consult with the typing judgements / rules on the slides!