CS4012 Topics in Functional Programming

Glenn Strong

Implementing Type Inference

As we know...

- Haskell is *strongly typed*, so incorrectly typed programs are rejected at compile time
  - Haskell is *polymorphically typed*, so types may be universally quantified over sets of type variables.
  - The Haskell compiler is capable of *automatic type inference* to discover the types of functions.

For completeness we note:

- Haskell functions can also take sets of types to provide a kind of structured overloading. This facility is provided via *type classes*
- and, of course, we know there are many extensions to the basic type system that people are experimenting with.
- We will ignore this today, but note that everything discussed today generalises with varying levels of ease to cover these

Today we will introduce some of the ideas behind Haskell's type inference mechanism.

The Haskell type checker must be able to

1. Determine whether the program is well typed
2. If so, determine the type of any expression in the program.

The programmer (presumably) believes their program is well typed. They have an idea for the type of *each expression* in the program. They could, in fact, have labelled each expression with that type.

In a sense, the job of the type checker is to recover the lost labels on the parts of the program.

The type inference system

Typically a type system is considered as two separate but related things:

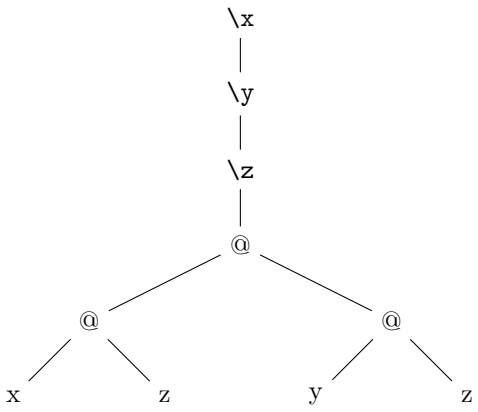
- A set of type, or *inference* rules whch give the logical framework for the system. This us usually used to demonstrate that the system is sound.
- An inference *algorithm* which can be used to deduce the types of expressions. Determining a usable inference algorithm for a given set of rules is usually non-trivial.

A small example

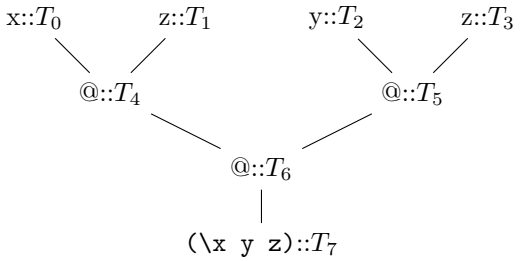
Take a simple expression:

```
\x y z -> x z (y z)
```

which we might lay out in a tree as:



We can decorate the tree with some labels, standing in for the types that we want to discover (we can call these *type variables*):



For convenience I've collapsed the lambda applications. I also inverted the tree because it will help us to line up notations later on.

For further convenience we will re-draw the tree in a notation that emphasizes the type variables (this notation will also be a help shortly in formalizing the rules).

$$\dfrac{\dfrac{\dfrac{x :: T_0 \qquad z :: T_1}{x\ z\ ::\ T_4}\ @ \qquad \dfrac{y :: T_2 \qquad z :: T_3}{y\ z\ ::\ T_5}\ @}{(x\ z)\ (y\ z)\ ::\ T_6}\ @}{(\lambda\ x\ y\ z)\ ::\ T_7}\ \lambda$$

For any sub-part of the tree shaped like this:

$$\dfrac{E_0 :: A \qquad E_1 :: B}{E_0\ E_1 :: C}\ @$$

- We know this is a function application (we can see that!),
- therefore the type of E_0 will contain an arrow (we know about functions)

We can write a *rule* for things of that shape:

$$\dfrac{E_0 :: t_0 \to t_1 \qquad E_1 :: t_0}{E_0\ E_1 :: t_1}\ \text{APP}$$

The label to the side of this rule lets us name it.

Applying the APP rule gives us some "type equations" from our tree:

$$\begin{aligned} T_0 &= \quad T_1 \to T_4 \\ T_2 &= \quad T_3 \to T_5 \\ T_4 &= \quad T_5 \to T_6 \end{aligned}$$

Substituting these back into the tree:

$$\dfrac{\dfrac{\dfrac{x :: T_1 \to T_4 \qquad z :: T_1}{x\ z\ ::\ T_5 \to T_6}\ @ \qquad \dfrac{y :: T_3 \to T_5 \qquad z :: T_3}{y\ z\ ::\ T_5}\ @}{(x\ z)\ (y\ z)\ ::\ T_6}\ @}{(\lambda\ x\ y\ z)\ ::\ T_7}\ \lambda$$

What to do with z? There are two type variables, T_1 and T_3 associated with the two instances of z.

Let's assume that T_1 and T_3 must be equal (in which case we can add these equations):

$$\begin{aligned} T_1 &= T_3 \\ T_7 &= T_0 \to T_2 \to T_1 \to T_6 \end{aligned}$$

The second equation comes from our knowledge of the meaning of lambda abstraction. We can write that in a rule as well:

$$\frac{x :: A \qquad E :: B}{\lambda\ x.E\ \ ::\ \ A \to B}\ \text{ABS}$$

Substituting back we would then get:

$$\dfrac{\dfrac{\dfrac{x :: T_1 \to T_4 \qquad z :: T_1}{x\ z\ \ ::\ \ T_5 \to T_6}\ @ \qquad \dfrac{y :: T_3 \to T_5 \qquad z :: T_3}{y\ z\ \ ::\ \ T_5}\ @}{(x\ z)\ (y\ z)\ \ ::\ \ T_6}\ @}{(\lambda\ x\ y\ z)\ \ ::\ \ (T_1 \to T_5 \to T_6) \to (T_1 \to T_5) \to T_1 \to T_6}\ \lambda\ x\ y\ z$$

Note that the ABS rule forced us to conclude that all occurrences of a lambda-bound variable have the same type.

We must make this a requirement if we are to be *sure* that the lambda abstraction will be usable in all contexts.

Type environments

We often know some facts about names that are used in the expressions. For example, in the expression:

$$\dfrac{\dfrac{map :: T_0 \qquad f :: T_1}{map\ f\ \ ::\ \ T_3}\ @ \qquad \dfrac{}{[1,2,3]\ \ ::\ \ T_4}}{map\ f\ [1,2,3]\ \ ::\ \ T_5}\ @$$

We actually know that (for example):

$$\begin{aligned} T_4 &= [Int] \\ T_0 &= (T_7 \to T_8) \to [T_7] \to [T_8] \end{aligned}$$

We represent the set of "known facts" about variables by a mapping, Γ, from names to types, which we call the "type environment".

All of the globally defined names in the program have entries:

$$
\begin{aligned}
1 &:: && Int \\
[1,2,3] &:: && Int \\
\text{map} &:: && (T_0 \to T_1) \to [T_0] \to [T_1]
\end{aligned}
$$

As we infer facts about variables and functions defined in the program we can add them to the type environment.

Type rules

The standard type rules refer to this environment. For example, here are some key rules:

$$\frac{}{\Gamma \cup \{x : t\} \vdash x : t}\ \textsc{Var}$$

$$\frac{\Gamma e : t' \to t \qquad \Gamma e' : t'}{\Gamma e\ e' : t}\ \textsc{App}$$

$$\frac{\Gamma \cup \{x : t'\} \vdash e : t}{\Gamma \vdash \lambda x.e : t' \to t}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash_p e : \sigma \qquad \Gamma \cup \{x : \sigma\} \vdash_p e' : \tau}{\Gamma \vdash_p \textbf{let}\ x = e\ \textbf{in}\ e' : \tau}\ \textsc{Let}$$

Worked example

Consider a small example language:

$E ::= c$	constant
$\mid x$	variable
$\mid \lambda\ x.E$	Abstraction
$\mid (E1\ E2)$	Application
$\mid let\ x\ =\ E_1\ in\ E_2$	let block

With a little language for types

$\tau ::= l$	base types
$\mid$t	type variable
$\mid \tau_0\ \to \tau_1$	Function types

And type schemes

$$\sigma ::= \tau$$
$$|\forall \mathrm{t}.\sigma_?$$

And type environments

$$T\ E ::= \mathit{Identifiers} \rightarrow \sigma$$

We are going to look at a complete implementation of a type inference engine for this little language. Before we dive into the code it's useful to take an overview

There are four main components to the type inference system:

- Types defining the AST of the expression language and of the type language
- Definitions that support the type environment
- Which includes definitions of *operations* on types and environments
- The implementation of the type inference rules
- The implementation of the *unification* algorithm which resolves type constraints

First, the AST for our language

```
data Exp = Var String
         | Const Val
         | App Exp Exp
         | Abs String Exp
         | Let String Exp Exp
     deriving (Eq,Ord)

data Val = I Integer | B Bool
     deriving (Eq,Ord)
```

Also, a small type language

```
data Type = TVar String
          | TInt | TBool
          | TFun Type Type
     deriving (Eq,Ord)
```

We also have parameterised type *schemes*

```
data Scheme = Scheme [String] Type
```

During type reconstruction we will make use of a type environment:

```
newtype TypeEnv = TypeEnv (Map.Map String Scheme)

remove                    ::  TypeEnv -> String -> TypeEnv
remove (TypeEnv env) var  =  TypeEnv (Map.delete var env)
```

We have the notion of type *substitutions*, which are mappings from type variables to types

```
type Subst = Map.Map String Type
```

For example:

```
nullSubst :: Subst
nullSubst = Map.empty
```

We need a function to find *free type variables* in a type expression, and a function to apply *type substutions*.

These can be applied to our types and to type schemes, so we make a class:

```
class Types a where
  ftv   :: a -> Set.Set String
  apply :: Subst -> a -> a
```

The apply function can be used, for example, when composing substutions:

```
composeSubst :: Subst -> Subst -> Subst
composeSubst s1 s2 = (Map.map (apply s1) s2) `Map.union` s1
```

```
instance Types Type where
    ftv (TVar n)      =  Set.singleton n
    ftv TInt          =  Set.empty
    ftv TBool         =  Set.empty
    ftv (TFun t1 t2)  =  ftv t1 `Set.union` ftv t2

    apply s (TVar n)      =  case Map.lookup n s of
                               Nothing  -> TVar n
                               Just t   -> t
    apply s (TFun t1 t2)  = TFun (apply s t1) (apply s t2)
    apply s t             = t
```

```
instance Types Scheme where
    ftv (Scheme vars t)
        =  (ftv t) `Set.difference` (Set.fromList vars)

    apply s (Scheme vars t)
        =  Scheme vars (apply (foldr Map.delete s vars) t)
```

For convenience we will extend these operations over lists as well:

```
instance Types a => Types [a] where
    apply s  =  map (apply s)
    ftv l    =  foldr Set.union Set.empty (map ftv l)
```

```
instance Types TypeEnv where
    ftv (TypeEnv env)      =  ftv (Map.elems env)
    apply s (TypeEnv env)  =  TypeEnv (Map.map (apply s) env)
```

```
4
5  generalize         ::  TypeEnv -> Type -> Scheme
6  generalize env t   =   Scheme vars t
7    where vars = Set.toList ((ftv t) `Set.difference` (ftv env))
```

We will need a way to supply fresh names whenever we want to invent new type variables during reconstruction.

The easiest way to do this is by supplying a state monad which can generate names, and run the type checker within this.

```
1  type TI a = ErrorT String (ReaderT TIEnv (StateT TIState IO)) a
2
3  data TIEnv = TIEnv  {}
4
5  data TIState = TIState {  tiSupply :: Int,
6                            tiSubst :: Subst}
```

```
1  runTI :: TI a -> IO (Either String a, TIState)
2  runTI t =
3      do (res, st) <- runStateT (runReaderT (runErrorT t)
4                                 initTIEnv) initTIState
5         return (res, st)
6    where initTIEnv = TIEnv {}
7          initTIState = TIState { tiSupply = 0,
8                                  tiSubst = Map.empty }
```

```
1  newTyVar :: String -> TI Type
2  newTyVar prefix =
3      do  s <- get
4          put s { tiSupply = tiSupply s + 1 }
5          return (TVar  (prefix ++ show (tiSupply s)))
```

The final preliminary is an instantiation function which replaces bound type variables with fresh new type variables

```
1  instantiate :: Scheme -> TI Type
2  instantiate (Scheme vars t) =
3       do  nvars <- mapM (\ _ -> newTyVar "a") vars
4           let s = Map.fromList (zip vars nvars)
5           return $ apply s t
```

Now that we have all the machinery in place we can begin inferring types...

```
1  tiConst :: TypeEnv -> Val -> TI (Subst, Type)
2  tiConst _ (I _)   = return (nullSubst, TInt)
3  tiConst _ (B _)   =  return (nullSubst, TBool)
```

More involved, as you'd expect, is the algorithm for inferring types for expressions

```
1  ti         ::  TypeEnv -> Exp -> TI (Subst, Type)
```

```
ti (TypeEnv env) (Var n) =
  case Map.lookup n env of
     Nothing    ->  throwError $ "unbound variable: " ++ n
     Just sigma ->  do t <- instantiate sigma
                       return (nullSubst, t)
```

```
ti env (Const l) = tiConst env l
```

```
ti env (Abs n e) =
 do tv <- newTyVar "a"
    let TypeEnv env' = remove env n
        env'' = TypeEnv (env' `Map.union`
                    (Map.singleton n (Scheme [] tv)))
    (s1, t1) <- ti env'' e
    return (s1, TFun (apply s1 tv) t1)
```

```
ti env (App e1 e2) =
    do  tv <- newTyVar "a"
        (s1, t1) <- ti env e1
        (s2, t2) <- ti (apply s1 env) e2
        s3 <- mgu (apply s2 t1) (TFun t2 tv)
        return (s3 `composeSubst` s2
                   `composeSubst` s1, apply s3 tv)
```

```
ti env (Let x e1 e2) =
    do  (s1, t1) <- ti env e1
        let TypeEnv env' = remove env x
            t' = generalize (apply s1 env) t1
            env'' = TypeEnv (Map.insert x t' env')
        (s2, t2) <- ti (apply s1 env'') e2
        return (s1 `composeSubst` s2, t2)
```

When inferring types we often needed to resolve potential conflicts. We deferred this to the `mgu` function.

This is the *unification* algorithm of Robinson.

```
mgu :: Type -> Type -> TI Subst
mgu (TFun l r) (TFun l' r')
   =  do  s1 <- mgu l l'
          s2 <- mgu (apply s1 r) (apply s1 r')
          return (s1 `composeSubst` s2)

mgu (TVar u) t               =  varBind u t
mgu t (TVar u)               =  varBind u t
```

```
mgu TInt TInt                =  return nullSubst
mgu TBool TBool              =  return nullSubst
mgu t1 t2
```

```
  =  throwError $ "types do not unify: " ++ show t1 ++
                  " vs. " ++ show t2

varBind :: String -> Type -> TI Subst
varBind u t
   | t == TVar u           =  return nullSubst
   | u `Set.member` ftv t  =  throwError $
                                "occur check fails: " ++ u ++
                                " vs. " ++ show t
   | otherwise             =  return (Map.singleton u t)
```

All done! We just need an entry point and we are good to go:

```
typeInference :: Map.Map String Scheme -> Exp -> TI Type
typeInference env e =
    do  (s, t) <- ti (TypeEnv env) e
        return (apply s t)
```