Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 6
The Implementation of Type Inference

CS4012
Topics in Functional Programming
Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

A worked example

As an illustration we will work out an *implementation* of the inference procedure.

To do this I am going to need a (very small!) functional language

$$
\begin{array}{ll}
E ::= c & \text{Constant} \\
\mid x & \text{Variable} \\
\mid \lambda\ x\,.\,E & \text{Abstraction} \\
\mid (E1\ E2) & \text{Application} \\
\mid let\ x\ =\ E_1\ in\ E_2 & \text{Let block}
\end{array}
$$

A worked example

I'm also going to need a *type* language, which comes in three parts. The language for basic types:

$$\begin{array}{lr} \tau ::= l & \text{Base types} \\ \mid \mathrm{t} & \text{Type variable} \\ \mid \tau_0 \rightarrow \tau_1 & \text{Function types} \end{array}$$

I also need a way to talk about *abstractions* of types; so-called "type schemes":

$$\begin{array}{l} \sigma ::= \tau \\ \quad \mid \forall \mathrm{t} \,.\, \sigma \end{array}$$

Finally, I have a type environment:

$$\Gamma ::= \text{Identifier} \rightarrow \sigma$$

A worked example

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:

1. **Types defining the AST of the expression language and of the type language**
2. **Definitions that support the type environment**
    1. **(which includes definitions of *operations* on types and environments)**
3. **The implementation of the type inference rules**
4. **The implementation of the *unification* algorithm which resolves type constraints**

The AST for the language can be translated quite directly:

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

type Name = String

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

We won't concern ourselves with details like parsing or even top-level definitions.

Part (2)

The type language

The type language is also quite direct

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

Type schemes could be thought of as ways to construct types with placeholder type variables:

```
data Scheme = Scheme [Name] Type
```

Part (2)

The type environment

During type reconstruction we will make use of an environment to keep track of information about names:

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

The environment can be extended and restricted, for example:

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

Part (2.1)

Operations on types and the environment

We have a notion of type *substitutions,* which perform the rôle of the equations we produced by hand earlier.

```
type Subst = Map.Map Name Type
```

The trivial substitution is the one with no information at all, but we will also make substitutions when we know one type variable could be replaced by another:

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

Part (2.1)

Operations on types and the environment

We will need a way to figure out what type variables are in use in a given type expression; that way we will know what names are in use in a particular type.

This is something we could usefully apply to types, type schemes, substitutions, and even lists of types (etc). So instead of making a single function we will use a class:

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

Part (2.1)

Operations on types and the environment

If you have two substitutions then you can combine them (basically, making a larger set of renaming out of two smaller ones). “apply” is needed here:

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

Part (2.1)

Operations on types and the environment

So for simple types the instances look like this:

```
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
```

Part (2.1)

Operations on types and the environment

If you have a type scheme then you have to remove any abstract variables from consideration:

```
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)
```

We can extend these operations over lists (it'll be handy):

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

Part (2.1)

Operations on types and the environment

And over type environments too.

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

Part (2.1)

Operations on types and the environment

If you want to include a type in the type environment then you want to include it in such a way as to let it be used in *any* context. Including one where some names already exist.

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

Part (2.1)

Operations on types and the environment

We are almost there.

The one other operation we did in our manual working of inference was to create lots of uniquely named type variables. We'll need a supply of fresh names, and the easiest way to get one is to run the type checker within a State monad that's tracking how many names we have already issued:

```
type TI a = State TIState a

data TIState = TIState {  tiSupply :: Int }
```

```
runTI :: TI a ->  (a, TIState)
runTI t = runState t initTIState
     where initTIState = TIState{ tiSupply = 0 }
```

Part (2.1)

Operations on types and the environment

This monad has two operations: one to make a fresh new type variable:

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

And one to turn a type *scheme* into a *type* (i.e. replace all the abstract variables with fresh names that don't already exist in this context):

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

Part (2.1)

Operations on types and the environment

This monad has two operations: one to make a fresh new type variable:

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

And one to turn a type *scheme* into a *type* (i.e. replace all the abstract variables with fresh names that don't already exist in this context):

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

Part 3

Implementing rules for inference

Now we have the machinery in place to begin inferring types

Every operation works on a type environment and something that can have a type. It computes a proposed type and a set of renamings.

Constants are simple

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

The more interesting operation is finding types for expressions.

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

Part 3

Implementing rules for inference

If you find a *name* then you need to look it up:

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

Constants can be expressions, by the way, so we'll quickly take care of those:

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

Part 3

Implementing rules for inference

Abstractions introduce new variables over the scope of their bodies:

```
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)
```

Part 3

Implementing rules for inference

Application is the now familiar App rule:

```
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)
```

This “mgu” operation is the “magic” that creates the necessary renamings.

Part 3

Implementing rules for inference

Local definitions finish off our language:

```
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)
```

Part 4

Resolving renamings

The “mgu” operation is called in when we have to figure out how to rename types to allow function applications.

```
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
```

Part 4

Resolving renamings

The “mgu” operation is called in when we have to figure out how to rename types to allow function applications.

```
mgu TInt TInt                 =  return nullSubst
mgu TBool TBool               =  return nullSubst
mgu t1 t2
  =  error $ "types do not unify: " ++ show t1 ++
                  " vs. " ++ show t2
```

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

A type inference system

We are done! We just need to run an expression through the system and see what types we get:

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

```
test e = runTI $ typeInference Map.empty e
```

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/