Type Inferencing in Functional Languages

Emily Ashworth, Tanya Bouman, Tonye Fiberesima
001402976, 001416669, 001231043
ashworel, boumante, fiberet

April 9, 2018

1 Introduction

Type inferencing is the process of analyzing type information in a program based on the use of some of its variables at compile time. [1] Many advanced languages, such as Swift [2] and Haskell [3], allow programmers to skip defining the types of variables, by doing type inferencing at compile time. Others, like Python, also allow the programmer to avoid defining the type of a variable, but these types are dynamic [4], and therefore not the topic of the current report, since this report focusses on compile time type inferencing. The most common type of type inferencing that languages use is the Hindley-Milner type system, which has two main implementations, Algorithm $\mathcal{W}$and Algorithm $\mathcal{M}$.

1.1 History

The original type inference algorithm was invented by Haskell Curry and Robert Feys in 1958 for the simply typed lambda calculus. In 1968, Roger Hindley worked on extending the algorithm and proved that it always produced the most general type. In 1978, Robin Milner solely developed an equivalent algorithm called Algorithm $\mathcal{W}$while working on designing ML and in 1985 Luis Damas proved that Milner's algorithm was complete and extended it to support polymorphic references. [5] Algorithm $\mathcal{M}$derived from

Algorithm $\mathcal{W}$was not formally presented until 1998 by Oukseh Lee and Kwangkeun Yi [6]. These algorithms are called the Hindley-Milner type inference algorithms, Algorithm W and Algorithm $\mathcal{M}$. Algorithm $\mathcal{W}$is the standard algorithm that works bottom to top which means that if a syntax tree were built from a piece of code, types would be inferred starting from the bottom while Algorithm $\mathcal{M}$traverse the syntax tree in the opposite direction of Algorithm $\mathcal{W}$. Types are inferred from the top of the syntax tree down.

2 Our Work

To do our work, we have created a toy language, and rules for that language, that demonstrate type inferencing with Algorithm $\mathcal{M}$. Inspiration for this comes from a paper demonstrating Algorithm $\mathcal{W}$in a similar manner. [7] We chose to use Algorithm $\mathcal{M}$because Algorithm $\mathcal{W}$is the more popular and more commonly used algorithm and also Algorithm $\mathcal{M}$always finds type errors earlier by considering a less number of expressions than Algorithm $\mathcal{W}$as proposed by Oukseh Lee and Kwangkeun Yi [6]. We would implement Algorithm $\mathcal{M}$with our toy language to demonstrate how type inferencing works in functional programming.

2.1 Our Toy Language

The toy language for the purpose of this project is a small portion of Haskell, allowing only the types `Int`, `Bool`, `String` and functions on those types. Later on in this report, we extend the language to also include type variables, but only a very limited use of them, as there are no type classes to constrain them.

2.2 Type Inferencing Rules

The type inferencing rules are rules specific to the language. These are used with Algorithm $\mathcal{M}$, Algorithm $\mathcal{W}$or any other type inferencing algorithm. Presented here are the rules necessary to infer types for our toy language. These rules are part of an existing set of language rules. [6].

The first rule is the constant rule, which simply states that a constant has an associated type.

(CON) $$\Gamma \vdash () : \iota$$

The next rule states that in order to know the type of the variable, we look it up from the context. This context or environment takes the form of a symbol table in a parser.

(VAR) $$\frac{\Gamma(x) \succ \tau}{\Gamma \vdash x : \tau}$$

Finally, there are two related rules for the definition and application of functions. For the definition of a function, there are two types, the type of the input and the type of the output. After adding the type of the input, e_1, to the context, we check the type of the output of the function, and make sure that it matches the given input and output type.

(FN) $$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

To apply the function, we need the type of the function and the argument. After finding the function type, we check that the argument matches the input type and return the output type of the function as the final result.

(APP) $$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

2.3 Inferencing

Let's assume that a parser has already produced a syntax tree, of type `Expression`, as given below. The `Type` that is given from the parser is the type given in the annotation. The annotation might or might not be correct. The type inferencing algorithm will detect that. The `Unknown` type is not part of the language and should not exist at the end of the type inferencing. The syntax tree given by the parser, however, often still contains `Unknown`'s.

```
data Type = TString
          | TInteger
          | TBool
          | Unknown
          | TFunc Type Type
          deriving (Eq, Ord)
```

```
data Expression = Var String Type
                | IntLiteral Int Type
                | StringLiteral String Type
                | BoolLiteral Bool Type
                | EFunc String Expression Type
                | Application Expression Expression Type
                deriving (Eq, Ord)

instance MonadFail (Either String) where fail = Left
```

Here is code that pretty prints code from the syntax tree. Pretty printing the language is necessary in order to give proper errors that relate back to the actual code. While the pretty print might not exactly match the program text, it will be close enough that the programmer can relate back to what they wrote. This also helps us for debugging, since it is easier to read than the expression type.

```
instance Show Expression where
  show (IntLiteral int typ) =
    case typ of
      Unknown -> show int
      _ -> show int ++ "␣::␣" ++ show typ
  show (StringLiteral str typ) =
    case typ of
      Unknown -> show str
      _ -> show str ++ "␣::␣" ++ show typ
  show (BoolLiteral bl typ) =
    case typ of
      Unknown -> show bl
      _ -> show bl ++ "␣::␣" ++ show typ
  show (Var str typ) =
    case typ of
      Unknown -> str
      _ -> str ++ "␣::␣" ++ show typ
  show (EFunc str exp typ) =
    let
      funStr =  "(\\" ++ str ++ "␣->␣" ++ show exp ++ ")
         "
    in
    case typ of
      Unknown -> funStr
      _ -> funStr ++ "␣::␣" ++ show typ
  show (Application e1 e2 typ) =
    case typ of
      Unknown -> show e1 ++ "␣" ++ show e2
      _ -> "(" ++ show e1 ++ "␣" ++ show e2 ++ ")␣::␣"
         ++ show typ

instance Show Type where
  show TString = "String"
  show TInteger = "Int"
  show TBool = "Bool"
  show Unknown = "?"
  show (TFunc t1 t2) = show t1 ++ "␣->␣" ++ show t2
```

3 Basic Type Inference

This is inference with $\mathcal{M}$. First we start with a helper function, `unify`. It would be a more interesting function if there were type variables, but for now, it just checks that the two types match.

```
unify :: Type -> Type -> Either String Type
unify Unknown t = Right t
unify t Unknown = Right t
unify (TFunc a b) (TFunc c d) =
  TFunc <$> (unify a c) <*> (unify b d)
```

```
unify s t =
  if s == t
  then Right s
  else Left $ "Error:␣could␣not␣match␣type␣" ++ show s
     ++ "␣with␣" ++ show t
```

Next, we get to the actual inference algorithm, as described by Lee and Yi. [6] In order to do inference with variables, we keep track of these in a basic symbol table which maps from the name of the variable to its type. This allows the definition of a variable in one place in the code and its use elsewhere. The process is similar for booleans and strings.

```
type TypeEnv = M.Map String Type
```

First we infer the types of constant literals. These are quite simple. For an integer literal, if the type signature given is an integer or if there is no type signature, we simply return integer as the type. Otherwise there is an error.

```
infer :: TypeEnv -> Expression -> Either String Type
infer env e@(IntLiteral i typ) = unify TInteger typ
infer env e@(BoolLiteral b typ) = unify TBool typ
infer env e@(StringLiteral s typ) = unify TString typ
```

Next we move on to the variable case. If the variable is defined and the type matches the current expected type, then we return that matching type. If the variable is not defined or the type does not match, there is an error.

```
infer env e@(Var v typ) =
  case M.lookup v env of
    Nothing -> Left ("Variable␣not␣in␣scope:␣" ++ show e
       )
    Just t -> unify t typ
```

Inferencing on functions is more interesting because the argument of the function needs to be made available in the type environment for the body of the function.

```
infer env e@(EFunc v exp typ) =
  case unify typ (TFunc Unknown Unknown) of
```

```
    Right (TFunc fin fout) ->
      TFunc fin <$> (infer (M.insert v fin env) exp)
    _ -> Left $ "Not␣a␣function:␣" ++ show e
```

For application inference, we find the type of the function and the argument, and then check that these two work together properly.

```
infer env e@(Application e1 e2 typ) = do
  e1type <- infer env e1
  e2type <- infer env e2
  (TFunc fin fout) <- unify e1type (TFunc Unknown typ)
  inType <- unify e2type fin
  unify typ fout
```

Here are examples.

1. A literal ‘5’ with type Int.

   ```
   let example1 = IntLiteral 5 TInteger
   let test1 = infer M.empty example1
   ```

   Unsurprisingly, this returns the type `TInteger`, since both the value of the literal and the type signature of the literal indicate that it is an integer.

2. A literal ‘5’ with type Bool.

   ```
   let example2 = IntLiteral 5 TBool
   let test2 = infer M.empty example2
   ```

   This, on the other hand, produces an error, because the value ‘5’ cannot have type `TBool`.

3. Now we move beyond checking whether or not the type signature is correct, to inferring a type when the signature is missing. Lets check with a literal ‘5’ and type Unknown

   ```
   let example3 = IntLiteral 5 Unknown
   let test3 = infer M.empty example3
   ```

   It produces the type TInteger which is the correct type for the literal 5

4. Also, let's consider inferring a type when we have a `TBool` literal 'True' with type Unknown

   ```
   let example4 = BoolLiteral True Unknown
   let test4 = infer M.empty example4
   ```

   As before, here it produces the type TBool which is the correct type for the literal True

5. Lastly in inferring a type, we can consider when we have a String literal with type Unknown

   ```
   let example5 = StringLiteral "Hello␣world" Unknown
   let test5 = infer M.empty example5
   ```

   It produces the type TString which is the correct type for the string literal "Hello World"

6. A string literal "Hi there" with type Bool

   ```
   let example6 = StringLiteral "Hi␣there" TBool
   let test6 = infer M.empty example6
   ```

   As expected this would produce an error because we have a string literal which is of type TString instead of TBool

7. A function with type Unknown. The Unknown type is where the type variables will become useful, since having Unknown types could be dangerous.

   ```
   let example7 = EFunc "x" (BoolLiteral False
      Unknown) Unknown
   let test7 = infer M.empty example7
   ```

8. A function application with type Unknown. Since the function body is the boolean false, the output type is false.

   ```
   let example8 = Application example7 (IntLiteral 5
      Unknown) Unknown
   let test8 = infer M.empty example8
   ```

9. The function here has the type `Unknown -> Unknown`. Since we are not yet supporting type variables, this result does not tell us that the two `Unknown`'s are the same, so further type inferencing in the next step will not tell us that the result of the application is an `Int`.

   ```
   let example9 = EFunc "x" (Var "x" Unknown) Unknown
   let test9 = infer M.empty example9
   ```

10. Here is where the trouble begins. Since `infer` does not know if the two Unknowns are the same or not, it is unable to determine that the result type should be an integer. This leads us to the next section, where we add type variables to that language, so that we can check whether or not these two `Unknown` types must be the same. This returns `Unknown`, but it should be a `TInteger`, since the input and output both have the type of the input, 10.

    ```
    let example10 = Application example9 (IntLiteral
       10 Unknown) Unknown
    let test10 = infer M.empty example10
    ```

4 Inference with Type Variables

In order to inference with type variables, we add the option `TVar` to the type. As indicated by the Lexical Structure section of the Haskell 89 report, the type variables must all begin with a lowercase letter. [8] The parser takes care of the enforcement, so we do not need to worry about this.

```
data Type = TString
          | TInteger
          | TBool
          | TVar String
          | Unknown
          | TFunc Type Type
          deriving (Eq, Ord)
```

The expression data type is exactly the same as before, since we do not add any new syntax.

However, the `Show` instance of the type will be different, since there is an added case for the type variable.

```
instance Show Type where
  show TString = "String"
  show TInteger = "Int"
  show TBool = "Bool"
  show (TVar v1) = v1
  show Unknown = "?"
  show (TFunc t1 t2) = show t1 ++ "␣->␣" ++ show t2
```

Now, we continue on again with `unify` and `infer`. This time `unify` is more interesting, because it has to deal with type variables. Only the added cases for type variables are shown.

```
unify :: Type -> Type -> Either String Type

unify (TVar s) t = Right t
unify t (TVar s) = Right t
```

The only case that changes for `infer` is the function case and the application, because there is now the option of giving the type `a -> a` or `a -> b` to a function. It is not necessary to consider type variables for variable expressions, since the type of a variable must be determined somewhere by a literal. The inference cases for literals are also the same as the previous ones.

Additionally, the type environment now much include information about type variables. Type variables are only defined within the context of a specific function, not for the entire program.

```
data TypeEnv = TypeEnv (M.Map String Type) (M.Map String
    Type)
```

To infer the type of a function, we get new type variables and assign those to the argument and output of the function. After unification, those types might still be variables, or they might be a specific type. Otherwise the inference continues in the same manner as before.

```
infer env@(TypeEnv typs vars) e@(EFunc v exp typ) =
  let
```

```
    nextVar :: String -> String
    nextVar (v:vs) =
      case M.lookup (v:vs) vars of
        Nothing -> v:vs
        _ -> nextVar ((succ v):vs)
    var1 = nextVar "a"
    var2 = nextVar var1
  in
  case unify typ (TFunc (TVar var1) (TVar var2)) of
    Right (TFunc fin fout) ->
      let
        newVars = M.insert var1 fin $ M.insert var2 fout
            vars
        newTypes = M.insert v fin typs
      in
      TFunc fin <$> (infer (TypeEnv newTypes newVars)
         exp)
    _ -> Left $ "Not␣a␣function:␣" ++ show e
```

Similarly, for the application of a function, the argument to the function might indicate that the output type is more specific, so we keep track of that in the type variable environment. The type variable environment becomes useful here for storing the new specific type of the variable and retrieving it later for use as the output type. First this function finds separately the types of the function and of the argument. After checking the compatibility of the input type, it checks if the argument made the function more specific. Note that the type of the function remains unchanged by applying it to a more specific argument. For example, a function of type `a -> a` still has the same type, even after being applied to an integer. Later on the program, it could be applied to a boolean. The more specific type of the function only exists for a particular application. Finally, it unifies the given argument with the expected output from the function.

```
infer env@(TypeEnv vs tvars) e@(Application e1 e2 typ) =
    do
  e1type <- infer env e1
```

```
(TFunc fin fout) <- unify e1type (TFunc Unknown typ)
e2type <- infer env e2
inType <- unify e2type fin
let tnew = case fin of
         TVar v -> M.insert v inType tvars
         _ -> tvars
case fout of
  TVar v ->
    case M.lookup v tnew of
      Just t -> unify typ t
      Nothing -> unify typ fout
  _ -> unify typ fout
```

Here is the code which runs the examples given below. It prints out each example and then the type of the entire example.

1. Again, we have our example of a identity function which takes and returns the same type. However, this time, `infer` recognizes that the input and output types must be the same, and gives them the same type variables.

   ```
   let example11 = EFunc "x" (Var "x" Unknown)
      Unknown
   let test11 = infer (TypeEnv M.empty M.empty)
      example11
   ```

2. This is again another function example from before, which now needs a type variable.

   ```
   let example12 = EFunc "x" (IntLiteral 5 Unknown)
      Unknown
   let test12 = infer (TypeEnv M.empty M.empty)
      example12
   ```

3. This is the example that caused trouble before. Now it can tell that the output must be an integer, since the argument is a `TInteger`.

```
let example13 = Application (EFunc "x" (Var "x"
   Unknown) Unknown) (IntLiteral 10 Unknown)
   Unknown
let test13 = infer (TypeEnv M.empty M.empty)
   example13
```

4. Making a function of two arguments in this toy language requires making a function which returns another function. In this case, the function looks like:

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

   in which `x` and `y` are both arguments.

   ```
   let example14 = EFunc "x" (EFunc "y" (Var "x"
      Unknown) Unknown) Unknown
   let test14 = infer (TypeEnv M.empty M.empty)
      example14
   ```

5 Conclusion

To conclude, type inferencing is a very interesting feature in the programming world. It can use Algorithm M, Algorithm W or other similar algorithm to infer the types of variables which are undeclared at compile time, thus enabling the programmer to be faster. Our example toy language shows how Algorithm M and the type inferencing system work. This only shows how type inferencing works for a very simple type system. However, the same kind of system can also do inference for a much more complicated type system, such as the one found in Haskell.

References

[1] D. Duggan and F. Bent, "Explaining type inference," *Science of Computer Programming*, vol. 27, no. 1, p. 37–83, 1996.

[2] "The swift programming language (swift 4.1) - types." Updated: 2018-02-20.

[3] "The haskell programming language - type inference." Modified: 29 November 2007.

[4] "What is python? executive summary." Accessed: March 21, 2018.

[5] K. D. LEE, *Fundations Of Programming Languages.* Springer International PU, 2018.

[6] O. Lee and K. Yi, "Proofs about a folklore let-polymorphic type inference algorithm," *ACM Trans. Program. Lang. Syst.*, vol. 20, pp. 707–723, July 1998.

[7] M. Grabmüller, "Algorithm w step by step,"

[8] "The haskell 98 report." Modified: December 2002.