Type Inferencing in Functional Languages

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

April 9, 2018

1 Introduction

Many advanced languages, such as Swift [1] and Haskell [2], 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 [3], and therefore not the topic of the currrent report. Languages that do type inferencing use the Hindley-Milner type system, which has two main implementations, Algorithm W and Algorithm M.

1.1 History

Algorithm W is standard algorithm, first done by (somebody), while Algorithm M was not formally presented until 1998 by Oukseh Lee and Kwangkeun Yi [4]. Algorithm W is an algorithm that works bottom to top. This means that if a syntax tree were built from a piece of code, types would be inferred starting from the bottom. Algorithm M is the exact inverse of Algorithm W. Types are inferred from the top of the syntax tree down.

2 Our Work

Because Algorithm W is the more popular and more commonly used algorithm, we decided to work with Algorithm M. To do our work, we have created a toy language, and rules for that language, that demonstrate type inferencing and Algorithm M.

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 these are not constrained by type classes.

2.2 Type Inferencing Rules

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.

```
-- these are the currently available types;
-- this might be expanded
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)
```

Pretty printing the language is necessary in order to give proper errors that relate back to the actual code.

```
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 $\mathcal{M}$

This 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 -> Maybe Type
unify Unknown t = Just t
unify t Unknown = Just t
unify (TFunc a b) (TFunc c d) =
  TFunc <$> (unify a c) <*> (unify b d)
unify s t =
  if s == t
  then Just s
  else Nothing
```

First we infer the types of constant literals. These are quite simple. 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.

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

```
infer :: TypeEnv -> Expression -> Either String Type
infer env e@(IntLiteral i typ) =
  case unify TInteger typ of
    Nothing -> Left $ "Type␣mismatch:␣literal␣" ++ show
       i
               ++ "␣cannot␣have␣type␣" ++ show typ
    Just i -> Right i
infer env e@(BoolLiteral b typ) =
  case unify TBool typ of
    Nothing -> Left $ "Type␣mismatch:␣literal␣" ++ show
       b
               ++ "␣cannot␣have␣type␣" ++ show typ
    Just b -> Right b
infer env e@(StringLiteral s typ) =
  case unify TString typ of
    Nothing -> Left $ "Type␣mismatch:␣literal␣" ++ show
       s
               ++ "␣cannot␣have␣type␣" ++ show typ
    Just s -> Right s
```

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 ->
      case unify t typ of
        Nothing -> Left ("Type␣mismatch:␣" ++ show e)
        Just s -> Right s
```

Inferencing on functions is more interesting because the argument of the function needs to be made available in the context for the body of the func-

tion.

```
infer env e@(EFunc v exp typ) =
  case unify typ (TFunc Unknown Unknown) of
    Nothing -> Left $ error ("Type␣mismatch:␣" ++ show e
       )
    Just (TFunc fin fout) ->  TFunc fin <$> (infer (M.
       insert v fin env) exp)

infer env e@(Application e1 e2 typ) =
  let
    -- infer the type of e1
    e1type = fromRight Unknown (infer env e1)
    e2type = fromRight Unknown (infer env e2)
  in
    -- unify that with what the output of the function
       should be
    case unify e1type (TFunc Unknown typ) of
      Just (TFunc fin fout) ->
        -- match the input type with the argument
        case unify e2type fin of
          Just t ->
            -- match the return type with the overall
               type
            case unify fout typ of
              Just s -> Right s
              Nothing -> error $ "Could␣not␣match:␣" ++
                 show fout ++ "␣and␣" ++ show typ
          Nothing -> error $ "Could␣not␣match:␣" ++ show
              e2type ++ "␣and␣" ++ show fin
      _ -> error $ "Could␣not␣match:␣" ++ show e1type ++
          "␣and␣" ++ show typ
```

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
```

   We can see that it have as the type Int which is the correct type for the literal 5

4. Also, lets infer a type with literal True and type Unknown

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

   We can see that it have as the type Bool which is the correct type for the literal True

5.
```
let example5 = StringLiteral "Hello␣world" Unknown
let test5 = infer M.empty example5
```

   Here we see that it have as the type String which is the correct type for the String literal Hello World

6. ```
   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 with type Bool.

7. ```
   let example7 = EFunc "x" (BoolLiteral False
      Unknown) Unknown
   let test7 = infer M.empty example7
   ```

8. ```
   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`.

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

10. ```
    example10 = Application example9 (IntLiteral 10
       Unknown) Unknown
    ```

However, the `Show` instance will be different.

```
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 much 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
```

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

```
infer env e@(EFunc v exp typ) =
  case unify typ (TFunc (TVar "a") (TVar "b")) of
    Right (TFunc fin fout) ->
      TFunc fin <$> (infer (M.insert v fin env) exp)
    _ -> Left $ "Not␣a␣function:␣" ++ show e
```

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

Here are examples.

1. ```
   let example15 = EFunc "x" (Var "x" Unknown)
      Unknown
   let test15 = infer M.empty example15
   ```

2. ```
   let example16 = EFunc "x" (IntLiteral 5 Unknown)
      Unknown
   let test16 = infer M.empty example16
   ```

4 Discussion

5 Conclusion

References

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

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

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

[4] 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.