Algorithm W Step by Step

Martin Grabmüller

Sep 26 2006 (Draft)

Abstract

In this paper we develop a complete implementation of the classic algorithm W for Hindley-Milner polymorphic type inference in Haskell.

1 Introduction

Type inference is a tricky business, and it is even harder to learn the basics, because most publications are about very advanced topics like rank-N polymorphism, predicative/impredicative type systems, universal and existential types and so on. Since I learn best by actually developing the solution to a problem, I decided to write a basic tutorial on type inference, implementing one of the most basic type inference algorithms which has nevertheless practical uses as the basis of the type checkers of languages like ML or Haskell.

The type inference algorithm studied here is the classic Algoritm W proposed by Milner [4]. For a very readable presentation of this algorithm and possible variations and extensions read also [2]. Several aspects of this tutorial are also inspired by [3].

This tutorial is the typeset output of a literate Haskell script and can be directly loaded into an Haskell interpreter in order to play with it. This document in electronic form as well as the literate Haskell script are available from my homepage[1]

This module was tested with version 6.6 of the Glasgow Haskell Compiler [1]

2 Algorithm W

The module we're implementing is called *AlgorithmW* (for obvious reasons). The exported items are both the data types (and constructors) of the term and type language as well as the function *ti*, which performs the actual type inference on an expression. The types for the exported functions are given as comments, for reference.

```
module AlgorithmW (Exp (..),
                   Type (..),
                   ti  -- ti :: TypeEnv → Exp → (Subst, Type)
                  ) where
```

We start with the necessary imports. For representing environments (also called contexts in the literature) and substitutions, we import module *Data.Map*. Sets of type variables etc. will be represented as sets from module *Data.Set*.

```
import qualified Data.Map as Map
import qualified Data.Set as Set
```

Since we will also make use of various monad transformers, several modules from the monad template library are imported as well.

[1] Just search the web for my name.

```haskell
import Control.Monad.Error
import Control.Monad.Reader
import Control.Monad.State
```

The module *Text.PrettyPrint* provides data types and functions for nicely formatted and indented output.

```haskell
import qualified Text.PrettyPrint as PP
```

2.1 Preliminaries

We start by defining the abstract syntax for both *expressions* (of type *Exp*), *types* (*Type*) and *type schemes* (*Scheme*).

```haskell
data Exp    = EVar String
            | ELit Lit
            | EApp Exp Exp
            | EAbs String Exp
            | ELet String Exp Exp
            deriving (Eq, Ord)
data Lit    = LInt Integer
            | LBool Bool
            deriving (Eq, Ord)
data Type   = TVar String
            | TInt
            | TBool
            | TFun Type Type
            deriving (Eq, Ord)
data Scheme = Scheme [String] Type
```

In order to provide readable output and error messages, we define several pretty-printing functions for the abstract syntax. These are shown in Appendix A.

We will need to determine the free type variables of a type. Function *ftv* implements this operation, which we implement in the type class *Types* because it will also be needed for type environments (to be defined below). Another useful operation on types, type schemes and the like is that of applying a substitution.

```haskell
class Types a where
  ftv   :: a → Set.Set String
  apply :: Subst → a → a
```

```haskell
instance Types Type where
  ftv (TVar n)      = {n}
  ftv TInt          = ∅
  ftv TBool         = ∅
  ftv (TFun t1 t2)  = ftv t1 ∪ 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
```

```haskell
instance Types Scheme where
  ftv (Scheme vars t)      = (ftv t) \ (Set.fromList vars)
  apply s (Scheme vars t) = Scheme vars (apply (foldr Map.delete s vars) t)
```

It will occasionally be useful to extend the *Types* methods to lists.

```haskell
instance Types a ⇒ Types [a] where
  apply s = map (apply s)
  ftv l   = foldr Set.union ∅ (map ftv l)
```

Now we define substitutions, which are finite mappings from type variables to types.

```haskell
type Subst = Map.Map String Type
nullSubst :: Subst
nullSubst = Map.empty
composeSubst       :: Subst → Subst → Subst
composeSubst s1 s2 = (Map.map (apply s1) s2) `Map.union` s1
```

Type environments, called Γ in the text, are mappings from term variables to their respective type schemes.

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

We define several functions on type environments. The operation $\Gamma\backslash x$ removes the binding for x from Γ and is called *remove*.

```haskell
remove                   :: TypeEnv → String → TypeEnv
remove (TypeEnv env) var = TypeEnv (Map.delete var env)
instance Types TypeEnv where
  ftv (TypeEnv env)     = ftv (Map.elems env)
  apply s (TypeEnv env) = TypeEnv (Map.map (apply s) env)
```

The function *generalize* abstracts a type over all type variables which are free in the type but not free in the given type environment.

```haskell
generalize       :: TypeEnv → Type → Scheme
generalize env t = Scheme vars t
  where vars = Set.toList ((ftv t) \ (ftv env))
```

Several operations, for example type scheme instantiation, require fresh names for newly introduced type variables. This is implemented by using an appropriate monad which takes care of generating fresh names. It is also capable of passing a dynamically scoped environment, error handling and performing I/O, but we will not go into details here.

```haskell
data TIEnv = TIEnv{ }
data TIState = TIState{tiSupply :: Int,
                       tiSubst :: Subst}
type TI a = ErrorT String (ReaderT TIEnv (StateT TIState IO)) a
runTI :: TI a → IO (Either String a, TIState)
runTI t =
  do (res, st) ← runStateT (runReaderT (runErrorT t) initTIEnv) initTIState
     return (res, st)
  where initTIEnv = TIEnv{ }
```

```
    initTIState = TIState{ tiSupply = 0,
      tiSubst = Map.empty }
newTyVar :: String → TI Type
newTyVar prefix =
  do s ← get
     put s{ tiSupply = tiSupply s + 1 }
     return (TVar (prefix ++ show (tiSupply s)))
```

The instantiation function replaces all bound type variables in a type scheme with fresh type variables.

```
instantiate :: Scheme → TI Type
instantiate (Scheme vars t) = do nvars ← mapM (λ_ → newTyVar "a") vars
                                 let s = Map.fromList (zip vars nvars)
                                 return $ apply s t
```

This is the unification function for types. The function *varBind* attempts to bind a type variable to a type and return that binding as a subsitution, but avoids binding a variable to itself and performs the occurs check.

```
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)
```

2.2 Main type inference function

Types for literals are inferred by the function *tiLit*.

```
tiLit :: TypeEnv → Lit → TI (Subst, Type)
tiLit _ (LInt _)  = return (nullSubst, TInt)
tiLit _ (LBool _) = return (nullSubst, TBool)
```

The function *ti* infers the types for expressions. The type environment must contain bindings for all free variables of the expressions. The returned substitution records the type constraints imposed on type variables by the expression, and the returned type is the type of the expression.

```
ti :: TypeEnv → Exp → TI (Subst, Type)
ti (TypeEnv env) (EVar n) =
  case Map.lookup n env of
    Nothing    → throwError $ "unbound variable: " ++ n
```

```
    Just sigma → do t ← instantiate sigma
                    return (nullSubst, t)
ti env (ELit l) = tiLit env l
ti env (EAbs 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 (EApp 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 (ELet 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)
```

This is the main entry point to the type inferencer. It simply calls *ti* and applies the returned substitution to the returned type.

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

2.3 Tests

The following simple expressions (partly taken from [2]) are provided for testing the type inference function.

```
e0 = ELet "id" (EAbs "x" (EVar "x"))
       (EVar "id")
e1 = ELet "id" (EAbs "x" (EVar "x"))
       (EApp (EVar "id") (EVar "id"))
e2 = ELet "id" (EAbs "x" (ELet "y" (EVar "x") (EVar "y")))
       (EApp (EVar "id") (EVar "id"))
e3 = ELet "id" (EAbs "x" (ELet "y" (EVar "x") (EVar "y")))
       (EApp (EApp (EVar "id") (EVar "id")) (ELit (LInt 2)))
e4 = ELet "id" (EAbs "x" (EApp (EVar "x") (EVar "x")))
       (EVar "id")
e5 = EAbs "m" (ELet "y" (EVar "m")
       (ELet "x" (EApp (EVar "y") (ELit (LBool True)))
         (EVar "x")))
```

This simple test function tries to infer the type for the given expression. If successful, it prints the expression together with its type, otherwise, it prints the error message.

```
test :: Exp → IO ()
test e =
  do (res, _) ← runTI (typeInference Map.empty e)
     case res of
       Left err → putStrLn $ "error: " ++ err
       Right t  → putStrLn $ show e ++ " :: " ++ show t
```

2.4 Main Program

The main program simply infers the types for all the example expression given in Section 2.3 and prints them together with their inferred types, or prints an error message if type inference fails.

```
main :: IO ()
main = mapM_ test [e0, e1, e2, e3, e4, e5]
```

This completes the implementation of the type inference algorithm.

3 Conclusion

This literate Haskell script is a self-contained implementation of Algorithm W [4]. Feel free to use this code and to extend it to support better error messages, type classes, type annotations etc. Eventually you may end up with a Haskell type checker...

References

[1] The GHC Developers. Glasgow Haskell Compiler Homepage. Available from: http://www.haskell.org/ghc, 2006. Last visited: 2006-06-14.

[2] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Sciences, Utrecht University, 2002.

[3] Mark P. Jones. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, 1999. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

[4] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

A Pretty-printing

This appendix defines pretty-printing functions and instances for *Show* for all interesting type definitions.

```
instance Show Type where
  showsPrec _ x = shows (prType x)
prType             :: Type → PP.Doc
prType (TVar n)    = PP.text n
prType TInt        = PP.text "Int"
prType TBool       = PP.text "Bool"
prType (TFun t s)  = prParenType tPP.⟨+⟩ PP.text "->"PP.⟨+⟩ prType s
```

```
prParenType   :: Type → PP.Doc
prParenType t = case t of
                  TFun _ _  → PP.parens (prType t)
                  _         → prType t

instance Show Exp where
  showsPrec _ x = shows (prExp x)

prExp                 :: Exp → PP.Doc
prExp (EVar name)     = PP.text name
prExp (ELit lit)      = prLit lit
prExp (ELet x b body) = PP.text "let" PP.⟨+⟩
                        PP.text x PP.⟨+⟩ PP.text "=" PP.⟨+⟩
                        prExp b PP.⟨+⟩ PP.text "in" PP.$$
                        PP.nest 2 (prExp body)
prExp (EApp e1 e2)    = prExp e1 PP.⟨+⟩ prParenExp e2
prExp (EAbs n e)      = PP.char '\\' PP.⟨+⟩ PP.text n PP.⟨+⟩
                        PP.text "->" PP.⟨+⟩
                        prExp e

prParenExp   :: Exp → PP.Doc
prParenExp t = case t of
                 ELet _ _ _  → PP.parens (prExp t)
                 EApp _ _    → PP.parens (prExp t)
                 EAbs _ _    → PP.parens (prExp t)
                 _           → prExp t

instance Show Lit where
  showsPrec _ x = shows (prLit x)

prLit            :: Lit → PP.Doc
prLit (LInt i)   = PP.integer i
prLit (LBool b)  = if b then PP.text "True" else PP.text "False"

instance Show Scheme where
  showsPrec _ x = shows (prScheme x)

prScheme                  :: Scheme → PP.Doc
prScheme (Scheme vars t)  = PP.text "All" PP.⟨+⟩
                            PP.hcat
                              (PP.punctuate PP.comma (map PP.text vars))
                             PP.<> PP.text "." PP.⟨+⟩ prType t
```