Regular Paper

Resolving Ambiguous Types in Haskell by Checking Uniqueness of Type Variable Assignments under Type Class Constraints

Yuya Kono[1,a] Hideyuki Kawabata[1,b] Tetsuo Hironaka[1,c]

Received: April 4, 2018, Accepted: August 8, 2018

Abstract: The type class mechanism, which introduces ad-hoc polymorphism into programming languages, is commonly used to realize overloading. However, this forces programmers to write many type annotations in their programs to resolve ambiguous types. Haskell's *type defaulting* rules reduce requirements for annotation. Furthermore, the widely used Glasgow Haskell Compiler (GHC) has an `ExtendedDefaultRules` (EDR) *extension* that facilitates interactive sessions so that the programmer avoids problems that frequently occur when using values like `[]` and `Nothing`. However, the GHC EDR extension sometimes replaces type variables with inappropriate types, so that, for example, the term `show . read` that is determined to have type `String -> String` under the GHC EDR extension does not exhibit any meaningful behavior because the function `read` in the term is considered to have type `String -> ()`. We present a flexible way of resolving ambiguous types that alleviates this problem. Our proposed method does not depend on default types defined elsewhere but rather assigns a type to a type variable only when the candidate is unique. It works with any type and type class constraints. The type to be assigned is determined by scanning a list of existing type class instances that meet the type class constraints. This decision is lightweight as it is based on operations over sets without using algorithms that require backtracking. Our method is preferable to using the GHC EDR extension since it avoids the use of unnatural type variable assignments. In this paper, we describe the details of our method. We also discuss our prototype implementation that is based on the GHC plugins, and the feasibility of modifying GHC to incorporate our method.

Keywords: type class, type ambiguity, type defaulting, Haskell

1. Introduction

The type class mechanism [9], [12] is used to introduce ad-hoc polymorphism into programming languages. It has been put into practical use for realizing overloading in the Haskell programming language [8]. For example, the function `show` in Haskell can be applied to arguments of different types. The type of `show` is represented with the type scheme `forall a. Show a => a -> String`. The left-hand side of `=>`, i.e., `Show a`, is called a type class constraint. It means that the type assignable to type variable `a` is limited to types that are instances of the type class `Show`. In the Haskell programming language, types `Integer` and `Bool` are defined as instances of type class `Show` with definitions of an overloaded function named `show`. Therefore, when the term `show True` is evaluated, the function `show` with a monomorphic type of `Bool -> String` is applied to the boolean value `True`. Similarly, for `show (42 :: Integer)`, the function `show` with a monomorphic type of `Integer -> String` is applied to the integer value `42 :: Integer`.

Introducing type classes causes ambiguous types [5], [7]. For example, the type of the function `read` in Haskell is represented with the type scheme `forall a. Read a => String -> a`. Therefore, the type of the composite function `show . read` is represented with the type scheme

```
forall a. (Show a, Read a) => String -> String.
```

That is, the *type* of `show . read` is determined to be `String -> String` independently of the type assigned to the type variable `a`, but we cannot determine the *behavior* of the composite function `show . read` until the type variable `a` is fixed because the behavior depends on the value of `a`. In Haskell, terms that have types represented by such ambiguous type schemes are not regarded as well-typed. If ambiguous types appear in a Haskell program, the programmer must write type annotations to enable determination of the types of type variables that cannot be fixed [8], [11].

It would not be a best solution to prohibit ambiguities because that would force programmers to write many type annotations in their programs. For instance, since numeric literals in Haskell have a polymorphic type represented by `forall a. Num a => a`, even if one wanted to simply use integer values, it might be necessary to attach a type annotation to each value that appears in the program such as `42 :: Integer`. Haskell's *type defaulting* rules alleviate this situation by assigning a default type declared in advance to each ambiguous type variable under certain conditions,

[1] Hiroshima City University, Hiroshima 731–3194, Japan
[a] ykono@ca.info.hiroshima-cu.ac.jp
[b] kawabata@hiroshima-cu.ac.jp
[c] hironaka@hiroshima-cu.ac.jp

which reduces the need to annotate types in programs [8].

The widely used Glasgow Haskell Compiler (GHC) [1] offers, in addition to type defaulting, an `ExtendedDefaultRules` (EDR) *extension* [3] that enables defaulting of ambiguous type variables under a less restricted condition. This promotes the usability of polymorphic constants, especially in interactive environments. When the EDR extension is enabled, some value constructors of polymorphic algebraic data types, such as `[]` and `Nothing`, become usable without type annotations. For example, terms such as `show []` are judged as well-typed in the interactive environment (GHCi) of GHC if this extension is enabled.

However, the GHC EDR extension allows its typechecker to assign unnatural types to ambiguous type variables. For example, when above-mentioned composite function `show . read` is typechecked, the unit type `()` is assigned to the ambiguous type variable, and the functions `read` and `show` are judged to have types `String -> ()` and `() -> String`, respectively. As a result, applying the composite function `show . read` of type `String -> String` to a string value of type `String`, which is obviously not a value of the unit type, causes a runtime error. This behavior of the composite function `show . read` cannot be predicted from its type; enabling the EDR extension of a typechecker could cause runtime errors that are difficult for the programmer to locate and remove.

Since the EDR extension can cause runtime errors, it is not suitable for batch compilation. We addressed this problem by developing a method for resolving type ambiguities that provides a more flexible way to write programs supporting user-defined types and batch compilation. It avoids causing runtime errors in ordinary situations by not assigning inappropriate types to type variables.

The contributions of the paper are summarized as follows:

(1) We present a method for resolving type ambiguities in a natural way without forcing the programmer to write either type annotations or additional rules related to types. The proposed method determines which types to assign to type variables by scanning the existing type class instances that meet the type class constraints, so that any well-typed terms can be evaluated without causing runtime errors. The type variable assignment determination is lightweight as it is based on operations over sets without using backtracking.

(2) We demonstrate the feasibility of the proposed method by presenting its implementation based on Jones' reference implementation of the Haskell typechecker [6].

(3) The prototype implementation of the method using GHC plugins is described. Although its functionality is limited due to the use of GHC plugins, its implementation demonstrated its effectiveness against existing large-scale Haskell programs.

The rest of the paper is organized as follows. In Section 2, we describe the type classes of the Haskell programming language and the problems related to type ambiguity. We also describe the type defaulting rules introduced into Haskell for avoiding ambiguous types. In Section 3, we describe the GHC `ExtendedDefaultRules` extension in detail and discuss its importance and related problems. In Section 4, we illustrate the type ambiguity problems that are difficult to be avoided when using existing methods and describe the method we developed for alleviating such problems. In Section 5, we describe how we implemented our proposed method in GHC. In Section 6, we discuss our method from several viewpoints. We conclude in Section 7 by summarizing the key points and mentioning future work.

2. Ambiguous Types and Type Defaulting

A type that is represented by a type scheme in which the type variables that have been introduced by universal quantifiers do not appear in the body of the type is called an *ambiguous type* [5], [7]. For example, the type of the term `show 42` is represented as

```
show 42 :: forall a. (Show a, Num a) => String.
```

The type of this term is ambiguous since the type variable `a` does not appear in the body of the type on the right-hand side of =>. In this paper, we call type variables to which type assignment cannot be performed, like `a` in the above example, *ambiguous type variables*.

In Haskell, numeric literals have polymorphic types. For example, the literal `42` has a type represented as

```
42 :: forall a. Num a => a,
```

and the type of `3.14` is represented as

```
3.14 :: forall a. Fractional a => a,
```

where type class `Fractional` is a subclass of type class `Num`.

Ambiguous types frequently appear in an environment in which there are many polymorphic type terms with type class constraints. In fact, ambiguous types related to type class `Num` tend to frequently appear in many programs. A Haskell programmer can omit type annotations for numeric literals because Haskell provides *type defaulting* rules for resolving ambiguous types by consulting a predefined list of declared default types for module.

Type defaulting lets the typechecker search a given list of default types for an appropriate type to assign to a type variable, say `v`, if the following conditions hold (Ref. [8], Section 4.3.4):

(1) All type class constraints in which `v` appears are in the form `C v` (where `C` denotes a type class).

(2) At least one of these type classes is `Num` or a subclass of `Num`.

(3) All of these type classes are defined in the `Prelude` module or in the standard Haskell libraries.

The list of default types can be defined separately in each module with a *default declaration*. For modules that do not have a default declaration, the types used as defaults are `Integer` and `Double`, in that order.

Type defaulting in Haskell reduces the need for explicit type annotations for resolving ambiguities due to the use of type class `Num`. For example, applying type defaulting disambiguates the type of the aforementioned term `show 42` to be `String`, with the type variable `a` assigned to be the type `Integer`.

In Haskell, type defaulting prevents some problems caused by type ambiguity while keeping numeric literals polymorphic. For example, if there is no type defaulting, a simple term `print $ 3.14 * 5 * 4` does not typecheck without a type annotation specifying the type of the subterm `3.14 * 5 * 4` given by the programmer. However, as mentioned above, the applicability of type defaulting is limited; all type classes appearing in the type class constraints of ambiguous types must be defined in the `Prelude` module or in the standard libraries. Therefore, it is not possible to resolve an ambiguous type in which user-defined type classes appear in its type class constraints. In addition, type defaulting covers only ambiguous types with type class constraints containing `Num` or its subclasses. Thus, type defaulting cannot resolve ambiguities caused by non-numeric polymorphic values like `[]` or `Left True`.

3. GHC's `ExtendedDefaultRules` Extension and Related Problems

3.1 `ExtendedDefaultRules` (EDR) Extension

The `ExtendedDefaultRules` *extension* (EDR) in GHC [3] relaxes the type defaulting conditions, thereby improving convenience in interactive environments. In an interactive environment, there are many situations in which values that have polymorphic types, such as `[]`, `Nothing`, and `Left "foo"`, are displayed on the screen. In those cases, an attempt is made to convert a value into a string by using a function like `show`. However, because polymorphic values are inherently ambiguous, we are often required to specify type annotations for those terms, that is, a tiresome requirement in interactive sessions. Use of the EDR extension alleviates this problem. By using the EDR extension, GHC's typechecker judges that terms `[]`, `Nothing`, and `Left "foo"` have types `[()]`, `Maybe ()`, and `Either String ()`, respectively.

When the EDR extension is enabled, the typechecker attempts to resolve ambiguous types by using the following procedure (Ref. [3], Section 4.4.8):

(1) Remove constraints that are not in the form `C a`, where `C` and `a` denote a type class and a type variable, respectively, and divide up the remaining set of constraints into groups such that all type class constraints in a group have an identical type variable.

(2) Among the divided groups of constraint sets, remove those sets that do not include any of the following type classes: `Num`, subclasses of `Num`, `Show`, `Eq`, `Ord`, `Foldable`, and `Traversable`.

(3) For each group, check whether each type on the list of default types satisfies all type class constraints in the group. Assign the first type that passes the check to the corresponding type variable.

In a preliminary step, the EDR extension places the unit type `()` and list type constructor `[]` at the head of the list of default types.

Figure 1 illustrates an example dialogue between the programmer and GHCi with the EDR extension enabled. On the first line, a type annotation is explicitly assigned to a polymorphic term `[]` as `[] :: String`. In this case, type defaulting does not work and the type of `show` is determined to be `String -> String`. On the third line, type defaulting by EDR is performed. The type of `[]` thereby becomes `[()]`, and the type of `show` becomes `[()] -> String`. Without the EDR extension, the programmer would have had to write the third line with an annotation like `show ([] :: [()])` in order to obtain the same result shown on the fourth line. Requiring the programmer to continually write such type annotations in an interactive environment would impose a great burden.

```
ghci> show ([] :: String)
"\"\""
ghci> show []
"[]"
```

Fig. 1 Example of interactive session in GHCi.

3.2 Problems with EDR Extension

Although the GHC EDR extension greatly improves convenience in GHCi, it can also cause confusions since the resultant assignments of types to ambiguous type variables are sometimes obscure. For example, since `show . read` has an ambiguous type, the term is judged to be ill-typed in standard Haskell. If the EDR extension is enabled, the type of the term is judged to be `show . read :: String -> String`. However, since the unit type `()` is assigned to the ambiguous type variable, the type of the function `read` in this composite function `show . read` is actually judged to be `String -> ()`. Therefore, applying the composite function `show . read` of type `String -> String` to a string value of type `String`, which is obviously not the unit type value, causes a runtime error. And compilers should not generate code that almost certainly will cause an error at runtime. However, under EDR, it is not possible to completely eliminate, by checking types, the chance of generating erroneous code. Thus, enabling the EDR extension could cause confusing bugs that are difficult to identify by simply "desk-checking" the source programs.

Note that, even if the EDR extension is enabled, there are still many cases in which the programmer must write type annotations. For example, even if a type that satisfies a set of particular type class constraints is uniquely found, the type cannot be assigned to an ambiguous type variable if it is not on the list of default types. The programmer must specify the type by writing a type annotation or modify the list of default types to include the type.

For these reasons, we do not expect that the use of an EDR extension is a best practice, especially when batch compilation is needed.

4. Resolving Ambiguous Types by Checking Uniqueness of Type Variable Assignments

4.1 Cases in which Type Defaulting and EDR are not Applicable

As described in Section 3, there are cases that cannot be properly handled by existing disambiguation methods. For example, the Haskell program shown in **Fig. 2** contains such an ambiguity. The type of the expression, `toString 42`, at the bottom is `forall a. (Num a, Outputable a) => String`, which contains an ambiguous type variable `a`. The type class `Outputable`

```
class Outputable a where
  toString ::  a -> String
  {- toString's definition omitted -}

instance Outputable Int where
instance Outputable Bool where

main = print $ toString 42
```

Fig. 2 Program with ambiguous type.

used in the example is a user-defined class. Haskell's standard type defaulting cannot deal with the ambiguity associated with `Outputable` because it tries to disambiguate only type variables related to type classes defined in module `Prelude` or in the standard libraries.

The GHC EDR extension has a wider range of applicable type class constraints compared to Haskell's type defaulting and can handle user-defined type classes. However, it relies on a predefined list of default types for disambiguation. For the program shown in Fig. 2, the type class `Outputable` has only two instances, i.e., `Int` and `Bool`. Although `Int` is also an instance of the type class `Num`, the expression `toString` in `toString 42` cannot be treated as having type `Int -> String` because `Int` is not on the list of default types.

Let us consider the situation from another perspective. Looking at the type class constraint `(Num a, Outputable a)`, we see that the type that is uniquely suited for assignment to the type variable `a` under this constraint is `Int`. Thus, a decision is made to assign `Int` to the ambiguous type variable `a`. Note that this decision does not require heavyweight processes. It is enough to check whether there is a unique type that is an instance of all type classes that appear in the type class constraints in order to determine whether there is a unique solution for a set of type class constraints. Doing this does not require a large amount of computational resources.

4.2 Proposed Method

Our proposed method resolves type ambiguities by assigning an ambiguous type variable a type that is a unique solution of the corresponding set of type class constraints. With this method, the type of the expression `toString 42` shown at the bottom of Fig. 2 is determined to be `String` by assigning `Int` to the ambiguous type variable.

Unlike existing methods such as type defaulting and EDR, the proposed method derives assignments for ambiguous type variables from type class constraints without resorting to predefined lists of default types. Also unlike type defaulting and EDR, our method does not place any restriction on type class constraints. Thus, it provides the programmer a much more flexible means of disambiguation, enabling the programmer to use any type and any type class constraint, including user-defined ones, in programs. The proposed method can be used with existing methods such as type defaulting and EDR, enabling them to avoid more type ambiguity problems.

Checking whether there is a unique assignment to a type variable `a` in a set of type class constraints ($\mathtt{C}_1$ `a`, $\mathtt{C}_2$ `a`, `...`, $\mathtt{C}_n$ `a`) can be done by counting the number of elements in the intersection of sets that contain instances of type class $\mathtt{C}_i$. The steps in disambiguation are summarized as follows:

(1) Partition type class constraints into groups so that the constraints in each group are associated with the same type variable; i.e., the type constraints in a group are of the form `C a`, where type variable `a` is identical.
(2) Construct a set of instances corresponding to each type constraint in each group in the current type class environment.
(3) Calculate the intersections among the sets of instances corresponding to type constraints in each group.
(4) For each group of constraints, if the previously calculated set contains exactly one type, assign it to the corresponding type variable `a`.

In the example program shown in Fig. 2, the last expression of `toString 42` has an ambiguous type represented by type scheme `forall a. (Num a, Outputable a) => String`. With our proposed method, the resultant intersection of type class constraints related to the ambiguous type variable `a` is {`Int`}. Thus, `Int` is assigned to `a` to eliminate the ambiguity. The proposed method thus solves the problem of type ambiguities by finding a unique assignable type at low cost.

5. Embedding Our Method in Haskell Typecheckers

Although there is no formal specification for Haskell's type inference, typecheckers are often implemented in Haskell compilers by incorporating a two-step constraint-based procedure: the first step generates the constraints and the second one solves them. For example, GHC's typechecker uses a constraint-based inference algorithm described in Ref. [11]. In Ref. [6], a typechecker of Haskell is implemented in Haskell. The reference implementation of the Haskell typechecker [6] is designed for readability and simplicity.

Here we briefly describe these representative typecheckers before introducing the implementation of our proposed method. We also discuss our prototype implementation that is based on the GHC plugins, and the feasibility of modifying GHC to incorporate our method.

5.1 Typechecking and Type Defaulting in Haskell

Constraint-based type inference resolves type ambiguities by first generating constraints and then solving them. In addition to type class constraints, the constraints include equational constraints and constraints on type families.

We illustrate the flow of a typical type inference procedure by using an example of typing a function `f` defined as

```
f x y = x == y.
```

(1) Assign type variables to partial terms

Type variables are assigned to terms as follows:

- `x == y ::` α
- `x ::` β
- `y ::` γ.

(2) Generate type constraints

The following constraints are generated:

- α ~ `Bool`
- $\beta \sim \gamma$

```
 1.     withUniqueCheck :: ([(Ambiguity, Type)] -> t) -> ClassEnv -> [Tyvar] -> [Pred] -> t
 2.     withUniqueCheck f ce vs ps = f (solve getinsts)
 3.       where vps = ambiguities ce vs ps
 4.             getinsts = do
 5.               amb@(_, preds) <- vps
 6.               let tss = do (IsIn i _) <- preds
 7.                            return (map (\(_ :=> IsIn _ t) -> t) $ insts ce i)
 8.               return (amb, tss)
 9.             solve xs = map (\(amb, tss) ->
10.                               (amb, head $ foldl1 intersect tss))
11.                        (filter (\(_, tss) ->
12.                                   single $ foldl1 intersect tss) xs)
13.             single [x] = True
14.             single _ = False
15.     ambiguities :: ClassEnv -> [Tyvar] -> [Pred] -> [Ambiguity]
16.     ambiguities ce vs ps = [ (v, filter (elem v . tv) ps) | v <- tv ps \\ vs ]
```

Fig. 3 Functions `withUniqueCheck` and `ambiguities` used to implement proposed method in the typechecker described by Jones [6].

- `Eq` β,

where ~ denotes a constraint that requires the types on both sides to be the same. For example, $\alpha \sim$ `Bool` means that α and `Bool` must be the same type.

(3) Solve type constraints

The following typing is obtained by solving the constraints:

`f :: forall a. Eq a => a -> a -> Bool`.

where the type variable `a` corresponds to the type variable β ($\sim \gamma$), which was not assigned a type in this step.

(4) Perform type defaulting

If the obtained type is ambiguous, type defaulting is performed. For GHC, disambiguation by EDR is attempted in this step if EDR is enabled. If some ambiguities still remain, the Haskell typechecker fails and an error is reported.

The proposed method is designed to be activated immediately before executing type defaulting. After types are obtained by solving the constraints, it attempts to disambiguate type variables, which is followed by type defaulting. Our method can thus be properly implemented along with type defaulting and EDR without their intervention.

5.2 Embedding Our Method in Typechecker of *Typing Haskell in Haskell*

Typing Haskell in Haskell by Jones [6] describes the details of implementing Haskell's typechecker in Haskell. The typechecker has been designed focusing on readability and simplicity and is thus appropriate to use in designing and prototyping the extended functionalities of Haskell's type system. It explains the functionalities used to deal with type ambiguity and explains type defaulting, along with its implementation [6] (Section 11.5.1).

The function `withUniqueCheck` in **Fig. 3** shows the major part of our implementation of the proposed method. This function accepts a function `f` that is used for disambiguation, a type class environment `ce`, a list `vs` of type variables that appear in the body of types and are not ambiguous, and a list of type class constraints `ps` of inferred types as arguments. The function `f` accepts a list of tuples of resolvable ambiguous type variables and types to be assigned as arguments. The function `withUniqueCheck` first determines the possibility of unique type variable assignments by following the procedure described in Section 4. Then, it applies the function `f` to the results to resolve type ambiguities by removing the type class constraints and registering tuples of type variables and types to the corresponding type environment. The behavior and the usage of the function `withUniqueCheck` resembles those of the function `withDefaults` [6], which manages type defaulting.

On Line 3 of the program shown in Fig. 3, a list `vps` of tuples of ambiguous type variables and corresponding type class constraints is computed by function `ambiguities`, which accepts a type class environment `ce` with instances, a list `vs` of type variables that appear in the bodies of types, and a list `ps` of type class constraints as arguments. Type class constraints related to ambiguous type variables are partitioned into groups so that each group is related to a single type variable.

The function `ambiguities` is identical to the function with the same name defined by Jones [6]. Function `tv` computes a list of type variables that appear in the type class constraints. Among the type variables listed in `tv ps`, those that do not appear in `vs` are treated as ambiguous type variables.

The value of `getinsts` defined on Line 4 in Fig. 3 is a list of instances defined in the environment `ce` for each type class in each group of type class constraints. Function `solve` defined on Line 9 computes the intersections among sets for each group and determines the assignments of types to ambiguous type variables by checking whether the number of elements in each intersection is 1. The type of the resultant value is a tuple of type `Ambiguity` and `Type`; `Ambiguity` represents a list of tuples of type variables and type class constraints, and `Type` represents a type.

Our proposed method can be implemented by inserting the overall code that performs disambiguation by using function `withUniqueCheck` immediately before the code that performs type defaulting in the typechecker [6].

5.3 Implementing Our Method in GHC

5.3.1 Implementation with Compiler Plugins

GHC offers a way to modify its functionality through APIs in the `ghc` package [*1]. In addition, GHC supports compiler plugins [3] for simple and plain modification of the behavior of the compiler. Functionalities implemented by using plugins are loaded into GHC at runtime. Programmers can add their own paths for doing something, e.g., resolving type constraints at type-

[*1] https://hackage.haskell.org/package/ghc

```
solve :: TcPluginSolver
solve _ _ [] = return $ TcPluginOk [] []
solve _ _ wanteds
 | all isValidCDictCan wanteds = do
     let splited = splitCDictCans wanteds
     xs <- mapM unifyCts splited
     let xs' = zip splited xs
               |> filter (isJust . snd)
               |> map (\(ct, Just tyeq) ->
                         (ct, tyeq))
     cts <- mkTyEqCts xs'
     return $ TcPluginOk [] cts
 | otherwise = return $ TcPluginOk [] []
 where splitCDictCans :: [Ct] -> [[Ct]]
       splitCDictCans xs =
         groupBy (\a b -> eqTypes
                          (cc_tyargs a)
                          (cc_tyargs b))
         $ sortBy (\a b -> nonDetCmpTypes
                           (cc_tyargs a)
                           (cc_tyargs b)) xs
       isValidCDictCan :: Ct -> Bool
       isValidCDictCan (CDictCan _ _ [_] _) = True
       isValidCDictCan _                    = False

mkTyEqCts :: [([Ct], (TyVar, Type))]
         -> TcPluginM [Ct]
mkTyEqCts xs = do
 xs' <- mapM mkTyEqCt xs
 return $ catMaybes xs'
 where mkTyEqCt :: ([Ct], (TyVar, Type))
                -> TcPluginM (Maybe Ct)
       mkTyEqCt (cts, (x, y)) =
         do b <- isTouchableTcPluginM x
            if b
              then Just
                   <$> newTyEqCt (head cts) x y
              else return Nothing

newTyEqCt :: Ct -> TyVar -> Type -> TcPluginM Ct
newTyEqCt ct ty1 ty2 = do
 let predty = mkTcEqPredLikeEv (ctEvidence ct)
              (mkTyVarTy ty1) ty2
 newWantedCt (ctLoc ct) predty
 where newWantedCt loc =
         fmap mkNonCanonical . newWanted loc

unifyCts :: [Ct] -> TcPluginM (Maybe (TyVar, Type))
unifyCts cs = do
 let t = cc_tyargs (head cs)
 env <- getInstEnvs
 let clsInstsList = map (getClsInsts env) cs
 let tyargList = map (map getTyArg) clsInstsList
 case (t, intersectsBy eqType tyargList) of
   ([x], [y]) | isTyVarTy x ->
                return (Just
                         (getTyVar "in gdt" x, y))
   _          -> return Nothing
```

Fig. 4 Implementation of proposed method by using GHC plugins.

checking and reconstructing expressions of intermediate representations for optimization, without modifying GHC itself.

We developed a prototype of our type disambiguator by using compiler plugins and confirmed that the extended typechecker described in Section 4 behaved as intended. The disambiguator determines whether there is a unique assignment to a type variable `a` following the steps described in Section 4. If there is a unique type `T`, another attempt is made to resolve the type constraints by using an additional type constraint, `a ~ T`.

Figure 4 shows the major functions of our implementation by using GHC plugins. When the GHC typechecker attempts to resolve type constraints, the additional function `solve` is invoked, which accepts a list of type constraints `wanteds` that are currently non-resolved as an argument. Function `unifyCts` accepts a list `cs` of to-be-solved type constraints and returns a list of tuples of uniquely assignable type variables obtained from `cs` and their corresponding types. Functions `mkTyEqCts` and `newTyEqCt` generate type constraints in the form `a ~ T` by using the list. Function `solve` computes a new list of type constraints by using those functions and returns it as the result of the processing by the compiler plugins.

```
class C1 a where
  f :: a -> String
instance C1 Int where
  f = show
instance C1 Bool where
  f = show
instance C1 Double where
  f = show

class C2 a where
  g :: a -> a
instance C2 Int where
  g = id
instance C2 Bool where
  g = id

main = print (f (g 42))
```

Fig. 5 Example program to be compiled with prototype implemented using compiler plugins.

Table 1 Elapsed times for compiling Haskell programs (s).

Application (Version)	pandoc (2.2)	hoogle (5.0.17.3)	hlint (2.1.5)
GHC8.4.1 with proposed method	366.8	39.7	37.5
GHC8.4.1	364.9	40.5	36.8

Testing using the prototype implementation of our method demonstrated that our proposed method can properly compile the example programs shown in Fig. 2 and **Fig. 5**. To evaluate the performance of the plugin-based implementation, we carried out a set of experiments in which three large-scale application programs written in Haskell (pandoc 2.2 [*2], hoogle 5.0.17.3 [*3], and hlint 2.1.5 [*4]) were compiled. Our prototype system correctly compiled all three applications. These results partially demonstrate that the additional functionality does not interfere with the behavior of the existing GHC typechecker.

Table 1 shows the elapsed times for compiling with and without our extension to the compiler. As shown in Table 1, the differences in compilation times due to plugin loading and other factors including excessive disambiguation of type variables, which will be described in Section 5.3.2, were negligible.

5.3.2 Beyond the Limitation: An Approach for Implementing Our Method in GHC

Implementation based on GHC plugins has inherently limited functionality: i.e., it is not possible to check whether the type class constraints in a particular set are causing type variable ambiguity or to check whether a particular ambiguity is resolvable without applying our method. Thus, the plugin-based implementation of our method assigns types to all type variables when there is a unique type class constraint "solution" even when the type variable is not actually ambiguous. This could result in making potentially polymorphic expressions monomorphic, which is not desired. For example, suppose a function is defined by using the type class `Outputable` shown in Fig. 2:

[*2] https://hackage.haskell.org/package/pandoc
[*3] https://hackage.haskell.org/package/hoogle
[*4] https://hackage.haskell.org/package/hlint

```
f x = toString (x + 1).
```

The type of the function `f` is expected to be `(Num a, Outputable a) => a -> String`. However, since there is no way to check whether the type variable `a` is ambiguous by using plugins, our prototype system will investigate the type class constraints, confirm the uniqueness of the type assignment to `a` in the environment, and then, infer that the type of function `f` is `Int -> String`.

To overcome this limitation when implementing our method in GHC, we must add a new typechecking path that is activated before carrying out type defaulting, as described in Sections 5.1 and 5.2. This cannot be done, however, without modifying the GHC source code.

In GHC version 8.4.1, resolving constraints and processing type defaulting are mainly done using the function `simplifyInfer` in compiler/typecheck/TcSimplify.hs*5. It should thus be possible to modify the GHC source code by revising only one file.

6. Discussion

There have been many studies on methods for resolving type ambiguities in Haskell [4]. The applicability of type defaulting in the standard implementation of Haskell is quite limited, and the assignments for ambiguous type variables are restricted to those related to type classes defined in `Prelude` or in the standard Haskell libraries. In contrast, GHC's EDR extension supports a wider range of type ambiguities, making the usage of Haskell more convenient in interactive environments. However, the intended type variable assignments can sometimes require an explicit declaration of a list of default types. As mentioned above, we consider the behavior of the GHC EDR extension to be inappropriate because the resultant assignments of type variables can seem unnatural to ordinary programmers. Several EDR extensions have been proposed [2]. Although they try to support the use of user-defined type classes for disambiguation, they use predefined lists of default types, which is different from our approach.

The reason for placing restrictions on the applicability of automatic type variable assignments by type defaulting is that the effect of type defaulting should be limited since the implicit approaches often cause unwanted assignments. In fact, GHC has a compile option (`-Wtype-defaults`) that causes warning messages to be displayed if type variable assignments are carried out by the type defaulting facility. Such warning messages would be helpful for maintaining the source code and figuring out unintended program behaviors. Compared to the implicit approaches, our proposed method for resolving type ambiguities does not use predefined lists of default types. Thus, using our method minimizes unintended assignments of type variables. Adding a facility for displaying warning messages would be also beneficial for our system.

We realize that our proposed method does not satisfy the claim by Vytiniotis et al. that "generalising over all constraints carries significant costs, and negligible benefits" [11]. They propose not using methods for assigning type variables that search for appropriate types because such a search could involve a huge number of existing instance declarations. Methods that search all existing type class instances in the environment often suffer an explosive increase in the cost for searching, especially in cases where there are many type variables. In addition, their use breaks the "open world assumption" of GHC's type system because the addition of instance declarations of type classes could destroy the once observed uniqueness of type variable assignments and affect the result of compilation, which is what they want to avoid. However, what they describe is a rather general typing strategy concerning ambiguities related to multi-parameter type classes [9] and type families [10]. In fact, our proposed method does not require a huge amount of computation. In addition, it is usable for common cases and can be implemented in a concise manner. Our approach thus does not contradict their claim. Many methods, e.g., EDR, type defaulting, and monomorphic restriction, have been implemented and utilized although their appropriateness was still unsettled [4]. Our approach may be in a similar position.

Most programming languages other than Haskell do not incorporate type class-like mechanisms for supporting the overloading of operators and values. The generics facility of Java and Rust require explicit type declarations when the values of type arguments are not determinable from the types of their arguments or results. Scala and OCaml support ad-hoc polymorphism in such a way that the problem of ambiguous types never occurs since type variables in type schemes always appear in their bodies.

7. Conclusion

We have described the problem of ambiguities in Haskell programs and have summarized two existing methods (type defaulting and the EDR extension of GHC) commonly used for resolving type ambiguities. We pointed out that there are cases in which type defaulting and the EDR extension cannot properly handle type ambiguities, and we presented a method for solving this problem. It does this by checking the uniqueness of type variable assignments under type class constraints.

We also presented an implementation of the proposed method as an extension of Jones' typechecker and presented results demonstrating its effectiveness. In addition, we discussed the use of our method to extend GHC's typechecker; presented how we built a prototype by using the GHC plugins, and confirmed the applicability of our approach to existing Haskell compilers.

Future work includes a detailed study introducing our disambiguation method, in an efficient and practical way, into systems that have, for example, multi-parameter type classes [9] and type families [10].

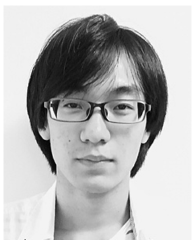
Acknowledgments The authors would like to thank Keisuke Nakano for his helpful comments. This work was supported in part by a KAKEN Grant-in-Aid for Scientific Research (C), No.17K00106.

References

[1] Glasgow Haskell Compiler home page, available from ⟨https://www.haskell.org/ghc/⟩ (accessed 2018-05-06).

*5 https://ghc.haskell.org/trac/ghc/browser/ghc/compiler/typecheck/TcSimplify.hs?rev=4df0106890df687d32ad1941dd5ccb31a11438d8

[2] GHC new feature request: Extending ExtendedDefaultRules, available from ⟨https://ghc.haskell.org/trac/ghc/ticket/8171⟩ (accessed 2018-05-06).
[3] GHC Team: GHC User's Guide Documentation, Release 8.4.1 (2018).
[4] Hudak, P., Hughes, J., Peyton Jones, S. and Wadler, P.: A History of Haskell: Being Lazy With Class, *3rd ACM SIGPLAN History of Programming Languages Conference* (*HOPL-III*), pp.12-1–12-55 (2007).
[5] Jones, M.P.: Coherence for qualified types, Research Report YALEU/DCS/RR-989, Yale University, New Haven, Connecticut, USA (1993).
[6] Jones, M.P.: Typing Haskell in Haskell, *Haskell Workshop* (1999), revised version available from ⟨http://web.cecs.pdx.edu/~mpj/thih/⟩ (accessed 2018-05-06).
[7] Jones, M.P.: Qualified Types: Theory and Practice, PhD Thesis, the University of Oxford (1992).
[8] Marlow, S. (Ed.): *Haskell 2010 Language Report* (2010).
[9] Peyton Jones, S., Jones, M. and Meijer, E.: Type classes: An exploration of the design space, *Haskell Workshop* (1997).
[10] Schrijvers, T., Peyton Jones, S., Chakravarty, M. and Sulzmann, M.: Type Checking with Open Type Functions, *Proc. International Conference on Functional Programming* (*ICFP '08*), pp.51–62 (2008).
[11] Vytiniotis, D., Peyton Jones, S., Schrijvers, T. and Sulzmann, M.: OutsideIn(X) – Modular type inference with local assumptions, *Journal of Functional Programming*, Vol.21, pp.333–412 (2011).
[12] Wadler, P. and Blott, S.: How to make ad-hoc polymorphism less ad hoc, *Proc. 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '89*), pp.60–76 (1989).



Yuya Kono is a student in the Department of Computer and Network Engineering, Hiroshima City University. His research interests include programming languages. He is a junior member of IPSJ.

Hideyuki Kawabata received B.E. and Ph.D. degrees from Kyoto University in 1992 and 2004, respectively. Since 2007, he has been a lecturer at Hiroshima City University. His research interests include numerical programming and programming languages. He is a member of ACM, IEEE Computer Society, IPSJ, IEICE, JSIAM, and JSSST.

Tetsuo Hironaka received a Ph.D. degree from Kyushu University in 1993. From 1993 to 1994, he served as a research associate at Kyushu University. From 1994 to 2006, he was an associate professor at Hiroshima City University. Since 2006, he has been a professor in the Computer Architecture Laboratory of Hiroshima City University. His research interests include computer architectures, reconfigurable architectures, and software engineering. He is a member of IPSJ, IEICE, IEEE, and ACM.