Assorted Types and a Type Class Default Mechanism for Type Ambiguities in Haskell

Koji Kagawa[1][0009−0001−8481−7493]

Kagawa University, Takamatsu Kagawa 761-0396, Japan
kagawa.koji@kagawa-u.ac.jp

Abstract. In functional languages, functions accepting a given data type can be defined freely, but constructors of the type cannot be added later. In object-oriented languages, adding subclasses to a given class is possible, but methods cannot be added to the original class. The difficulty of achieving extensibility in both directions is known as the expression problem. In Haskell, type classes mitigate the problem to some extent. That is, the types that methods (functions) can operate on can be added later. However, problems arise when grouping these target types into a single list or a similar structure. Several approaches have been proposed to address this, but issues remain, such as bad chemistry with types that include reference types. This paper proposes a mechanism for defining assorted types to group multiple types, along with a type class defaulting mechanism to handle the resulting type ambiguity.

Keywords: Haskell · type class · type ambiguity · type class defaulting mechanism.

1 Introduction

In functional programming languages, one can generally define as many functions as desired for a given data type, but the varieties of data (constructors) cannot be added later. Conversely, in object-oriented languages, one can freely increase the varieties of data (subclasses) for a given data type (class), but functions (methods) cannot be added to the original class. This well-known problem that it is difficult to achieve extensibility in both directions simultaneously is termed the expression problem [15].

However, in Haskell, type classes mitigate the issue to some extent. That is, while you cannot add new constructors for a function's target type, you can define functions that target multiple types, allowing the target types themselves to be added later. In this sense, Haskell functions can increase the varieties of data types passed as arguments. However, in this case, the differing types present another problem when attempting to store data of multiple types, which are the targets of the function, within a single container such as a list or an array. When using containers, it is necessary to ensure the elements are of the same type. Similarly, when using references, the types must be the same. This occurs when one wants to assign multiple types, sharing common operations, successively to

a single reference. Therefore, several techniques have been proposed to enable grouping different types belonging to the same class into a single container.

Generally, object-oriented programming is well-suited to imperative techniques using references. When performing an upcast (type conversion from subclass `b` to superclass `a`), some information—such as methods or fields (instance variables)—may become inaccessible. Without references, we would have to restore the lost information by performing a downcast (type conversion from superclass `a` back to subclass `b`) to use methods specific to the subclass. However, extracting a portion of data and then restoring it after update is essentially the same role as references. On the other hand, when references are used in the first place, the object's identity is preserved, allowing access to the lost information from a location separate from the container.

The expression problem is often viewed in functional programming languages as an issue encountered when handling data types such as abstract syntax trees used by compilers. However, resolving the expression problem is also crucial when emulating object-oriented programming in functional languages, and the solution should ideally be compatible with reference types.

The mechanism proposed in this paper possesses the following characteristics compared to existing proposals.

- It is well-suited to reference types.
  It can be used within references, or references can be used within data types. However, the technique proposed in this paper is still useful even when reference types are not involved.
- It requires no advance preparation.
  There is no need to define data types with the prior intention of using them with heterogeneous containers or reference types.
- It supports separate compilation.
  The data types provided by this proposal translate straightforwardly to existing type classes (with commonly used extensions such as `MultiParamTypeClasses` and `FunctionalDependencies`) and algebraic data type definitions. Furthermore, the processing required to consolidate multiple data types into a single container is not expensive.

The remainder of this paper is structured as follows: first, we briefly explain existing research on the expression problem in Haskell (Section 2), then go on to introduce the proposed mechanism of assorted type definition and type class defaulting scheme (Section 3). We also refer to other related research in Section 4. Finally, we outline future challenges (Section 5), and conclude (Section 6).

2 Background

This section explains several existing proposed methods for constructing heterogeneous containers composed of elements derived from multiple types.

2.1 Objects and Subtyping in a Functional Perspective

Odersky [11] proposes a technique for constructing heterogeneous containers of types sharing the same interface.

For example, consider the following type class C:

```
class C a where
    view :: a → Int → String
    handle :: a → String → a
```

which represents some document-like data types with methods view to display a portion of the document and handle to handle modifications to the document. To group types belonging to this C class into a single container, one may define a type T corresponding to a tuple of methods such as the following.

```
data T = T (Int → String) (String → T)
```

Here, the Int → String field corresponds to the view method, and the String → T field corresponds to the handle method.

This type T is an instance of the class C, as shown below, and any instance of C can be converted to T.

```
instance C T where
    view (T f g) n = f n
    handle (T f g) s = g s

toT :: C a ⇒ a → T
toT x = T (view x) (λs → toT (handle x s))
```

Generally, such a type T can be defined when all the method types of the type class C a are of the form a → τ, where a appears positively in the return type τ.

The limitation of this approach is that when the number of involved functions increases, the definitions of class C and type T must be modified. That is, once the types of heterogeneous containers are fixed, it is still possible to add new types afterwards (i.e., to add new instances of the C class), but it is not possible to add functions (methods) (e.g., gview :: a → Int → ByteString to display a portion of the document graphically) without recompilation. One could define a subclass C' of C to add new methods, but the corresponding type T' would also need to be newly defined.

A variation using existential types [7], as shown below, also exists, but the above restriction applies equally.

```
{-# LANGUAGE ExistentialQuantification #-}
data T = forall a. C a ⇒ T a (a → Int → String) (a → String → a)
```

2.2 Data Types à la Carte

Swierstra proposed a method in Data Types à la Carte [14] for combining multiple types into a single sum type. This is achieved by defining operators for the

type constructor :+:, as shown below, and defining recursive data types by tying the knot such as Expr.

The following example combines two types, Lit and Add, using the :+: operator.

```
data Expr f = In (f (Expr f))
data Lit e = Lit Int
data Add e = Add e e
data (f :+: g) e = Inl (f e) | Inr (g e)

addExample :: Expr (Lit :+: Add)
addExample = In (Inr (Add (Inl (Lit 2)) (Inl (Lit 3))))
```

Using constructors such as Inr and Inl directly is cumbersome, so one can construct data using the overloaded function inj.

```
class (Functor sub, Functor sup) ⇒ sub :<: sup where
    inj :: sub a → sup a
```

For example, one can define instances such as the following.

```
instance (Functor f, Functor g) ⇒ f :<: (f :+: g) where
    inj = Inl
```

Using this inj, one can define the following "smart constructors."

```
lit :: (Lit :<: f) ⇒ Int → Expr f
lit x = In (inj (Lit x))

add :: (Add :<: f) ⇒ Expr f → Expr f → Expr f
add x y = In (inj (Add x y))
```

Then, the above addExample can be rewritten as follows.

```
addExample :: Expr (Lit :+: Add)
addExample = add (lit 2) (lit 3)
```

When adding new functions to these types, one can define a type constructor class as a subclass of Functor.

```
class Functor f ⇒ Eval f where
  evalAlgebra :: f Int → Int

instance Eval Lit where
  evalAlgebra (Lit x) = x

instance Eval Add where
  evalAlgebra (Add x y) = x + y

instance (Eval f, Eval g) ⇒ Eval (f :+: g) where
  evalAlgebra (Inl x) = evalAlgebra x
  evalAlgebra (Inr y) = evalAlgebra y
```

Here, Functor is a type (constructor) class defined as follows.

```
class Functor f where
    fmap :: (a → b) → f a → f b
```

Meanwhile, functions for the Expr type can be defined using the following foldExpr function.

```
foldExpr :: Functor f ⇒ (f a → a) → Expr f → a
foldExpr f (In t) = f (fmap (foldExpr f) t)

eval :: Eval f ⇒ Expr f → Int
eval expr = foldExpr evalAlgebra expr
```

This approach also allows functions to be added later by defining classes such as Eval, and new type categories like Lit or Add can be also introduced. These types can then be packaged into containers using inj.

However, data types à la carte also present several drawbacks. Firstly, it requires even non-recursive constructors to be defined as type constructors that take a type parameter corresponding to their own type (e.g., e for the Lit type above). This is at least unnatural and necessitates planning in advance for combining with other data types later. Furthermore, **each constituent type constructor** must be a Functor, meaning it must be able to define fmap, and notably cannot contain a reference to its own type. However, this is unnatural, as one should ideally be able to simply "forget" about methods that update self-referential types. In contrast, Odersky's approach above only requires **each method return type** to be a Functor.

2.3 Typed Tagless Final Interpreters

Kiselyov's "Typed Tagless Final Interpreters" (TTFI) [6] is a technique for representing data types not by defining algebraic data types, but by defining corresponding type classes. For example, corresponding to the above Lit and Add, we define the following type class:

```
class ExpSYM repr where
    lit :: Int → repr
    add :: repr → repr → repr

tf1 = add (lit 8) (add (lit 1) (lit 2))
```

Furthermore, when adding functions to a data type, we define instances for the previously described type class as follows.

```
instance ExpSYM Int where
    lit n = n
    add e1 e2 = e1 + e2

eval :: Int → Int
eval = id
```

Evaluating eval tf1 yields 11. For example, the type of tf1 is ExpSYM repr ⇒ repr. By defining a subclass of ExpSYM, one can add constructors, and by defining an instance of ExpSYM, one can add functions.

However, this approach, representing data as polymorphic functions, is ill-suited for uses such as passing functions as arguments or storing them in references. As is known from ML, references have restrictions on the generalisation

of type variables. That is, one cannot put a type like $\forall$ repr. ExprSYM repr $\Rightarrow$ repr into a reference; one must instantiate the type variable instead. ML has restrictions known as the value restriction [16]. In Haskell, using functions like newSTRef or newIORef ensures their return values are lambda-bound [12]. In the TTFI approach, restricting the type to monomorphic means only one method can be used. The same paper [6] proposes a method using cloning (duplicating) to enable multiple methods, but this undeniably complicates the program, and it is questionable whether this technique can be applied to data containing references.

2.4 Open Data Types and Open Functions

Löh *et al.* proposed a syntax for Haskell in "Open Data Types and Open Functions" [8]. They also proposed candidate implementation methods. The proposed syntax uses a keyword open and is as follows.

```
open data Expr :: *

Lit :: Int → Expr
Add :: Expr → Expr → Expr

open eval :: Expr → Int
eval (Lit n) = ···
```

However, this proposed syntax has not been incorporated into the Haskell standard.

Reasons for this may include the implementation making separate compilation difficult. Though they propose a translation scheme to maximise separate compilation, still it requires recompilation of a small part of the program when adding new constructors. And since it modifies the meaning of existing type identifiers, say Expr, it is necessary to resort to recursive modules to allow cyclic dependencies between modules. This would complicate import statements by necessitating the use of explicit module *interfaces*.

2.5 Polymorphic Variants

In OCaml, polymorphic variants are defined using backquote character (‘) as follows. (Ocaml uses semicolon (;) to separate list items instead of comma (,) as in Haskell.)

```
let exps1 = [`Lit 1; `Add(`Lit 2, `Lit 3)];;
```

This exps1 has a type [> ‘Add of [> ‘Lit of int] * [> ‘Lit of int] | ‘Lit of int] list. Here, the symbol > means that the type can have more constructors than those listed, and therefore it is possible to add other constructors freely. On the other hand, if we define functions for this type straightforwardly as follows,

```
let rec eval = function
      | `Lit n → n
      | `Add (e1, e2) → eval e1 + eval
    ;;
```

this eval has type ([< 'Add of 'a * 'a | 'Lit of int] as 'a) -> int. Here, the symbol < means that the type can have fewer constructors than those listed, meaning the function cannot accept other constructors and is not extensible.

Garrigue proposed using open recursion to define functions accepting more constructors [4].

```
let eval_add eval_rec = function
    | `Lit n -> n
    | `Add(e1, e2) -> eval_rec e1 + eval_rec e2
;;
let rec eval1 e = eval_add eval1 e;;
```

Then we can define another function eval2 that accepts more constructors by reusing eval_add.

```
let eval_add_mult eval_rec = function
    | (`Lit _ | `Add _) as e -> eval_add eval_rec e
    | `Mult(e1, e2)  -> eval_rec e1 * eval_rec e2
;;
let rec eval2 e = eval_add_mult eval2 e;;
```

On the other hand, as OCaml lacks type classes, a different mechanism such as first-class modules would be required to use the same name for functions accepting different sets of constructors.

As for Haskell, the author previously proposed extensions to introduce polymorphic variants [5]. However, this required adding new syntax for defining polymorphic variants and for declaring instances between polymorphic variants and type classes, alongside modifications to the type inference algorithm, making its incorporation into Haskell's standard difficult. The mechanism we propose in this paper can be considered a restructuring that organises the situations where variant polymorphism manifests, enabling its implementation using Template Haskell's quasi-quotation [13] and GHC's compiler plugins.

3 Proposed Mechanism

3.1 Working Example

In the following, we will use several types and methods as concrete examples. These are the data types and methods used in turtle graphics.

```
data Turtle s = Turtle { x :: STRef s Double
                       , y :: STRef s Double
                       , t :: STRef s Double }

class Movable s a | a → s where
    forward :: a → Double → ST s ()
    turn ::    a → Double → ST s ()
```

We will subsequently add more data types and methods, and examine examples where these are grouped together for handling using containers or references.

For instance, we consider adding a color field as shown below,

```
data ColorTurtle s = ColorTurtle { x:: STRef s Double
                                 , y:: STRef s Double
                                 , t:: STRef s Double
                                 , c:: STRef s Int }

class HasColor s a | a → s where
  getColor :: a → ST s Int
  setColor :: a → Int → ST s ()
```

or introducing turtles dependent on other turtles as follows.

```
newtype ContraryTurtle s e = ContraryTurtle (STRef s e)

instance Movable s e ⇒ Movable s (ContraryTurtle s e) where
    forward (ContraryTurtle r) d = readSTRef r >>= λt → forward t d
    turn (ContraryTurtle r) a    = readSTRef r >>= λt → turn t (−a)
    -- Contrary turtle moves in the direction opposite to the told direction.
```

Here, e denotes a type variable representing the type of another turtle.

3.2 Basic Idea

The fundamental idea underlying the proposal in this paper is, in a sense, straightforward. Namely, we define an algebraic data type that possesses all the types one wishes to include in the container as its constituent elements. We define a recursive data type by constructing a sum type and, if necessary, defining the corresponding parts of the subtree to be of the same type as the type itself. That is, we tie the recursive knot. We refer to such a type, which takes a sum and further ties the necessary recursive knot, as an **assorted type** in this paper. However, this proposal does not require, unlike data types à la carte, that each component type of the sum be a Functor over itself. Consequently, generalised type constructors like Expr or :+: cannot be provided, necessitating a new algebraic data type for each set of components.

However, this means the definition of the algebraic data type must be modified whenever the type of a component is increased. This point is the opposite of Odersky's approach. That is, functions can be added even after creating a heterogeneous container, but once the type is fixed, new constituent types cannot be added. Therefore, to delay fixing the definition of the algebraic data type, the constructors are made anonymous, and data is constructed using functions overloaded on the return type. Such usage of overloading to delay type determination is also common to data types à la carte and TTFI. However, unlike TTFI, which represents data as polymorphic functions, the proposed mechanism represents data as algebraic data types, making it suitable for use with references.

3.3 Definition of Assorted Type

When defining an assorted type, each constructor is composed of type conversions from other types, hence there can be only one field. This aspect is analogous to newtype. Furthermore, multiple constructors are permitted, but constructor

names are not specified. This aspect is the dual of newtype. It employs syntax similar to the standard data declaration, but uses the new keyword assorted.

```
data assorted AllTurtle s = Turtle s
                          | ColorTurtle s
                          | ContraryTurtle s (AllTurtle s)
    deriving (Movable s _, HasColor s _)
```

Note that Turtle, ColorTurtle, and ContraryTurtle here are not constructor names, but type constructor names of the constituent types already defined. In practice, it is implemented using Template Haskell Quasi-quotation, meaning it is written within the ... portion of the quasi-quotation `[assorted| ... |]`.

The grammar for defining assorted types is generally as follows: Here, *tycon*, *type*, etc., follow the names of non-terminal symbols used in the extended BNF of the Haskell specification [9]. (for simplicity, parts such as module names are ignored.)

```
data assorted tycon tyvar_1 ... tyvar_k = type_1 | ... | type_m
    deriving (class_1, ... class_n)
```

The deriving clause serves to declare instances for this type. For a standard data declaration, the deriving clause lists the type class names, such as (Eq, Show). However, assorted types typically possess type parameters, meaning the type classes in the deriving clause are necessarily multi-parameter. Only one type parameter within it is independent; the other type parameters depend on that independent parameter. Therefore, we extend the deriving notation: we use "_" to denote the (independent) parameter corresponding to the data type being defined within the class expressions. The remaining dependent type parameters may also include the type parameters of the defined type constructor ($tyvar_1$, ..., $tyvar_k$).

The above AllTurtle translates to the following data type definition and instance declarations. Here, D1, D2 and D3 are fresh constructor names that are not visible outside.

```
data AllTurtle s = D1 (Turtle s)
                 | D2 (ColorTurtle s)
                 | D3 (ContraryTurtle s (AllTurtle s))

instance Cast (Turtle s) (AllTurtle s) where
    ucast = D1
    dcast (D1 t) = Just t
    dcast _ = Nothing

instance Cast (ColorTurtle s) (AllTurtle s) where
    -- Omitted, as it is largely similar.

instance Cast (ContraryTurtle s (AllTurtle s)) (AllTurtle s) where
    -- Omitted
```

Here, the Cast a b class is defined as follows.

```
class Cast a b where
    ucast :: a → b
    dcast :: b → Maybe a
```

Furthermore, for the classes within the deriving clause, AllTurtle is declared as their instances by utilising the instances of each component's data type.

```
instance Movable s (AllTurtle s) where
    forward (D1 t)  = forward t
    forward (D2 ct) = forward ct
    forward (D3 tt) = forward tt
    ...
```

Such methods may only be defined when each method's type is of the form $\mathsf{a} \to \tau$, where a appears positively in τ. Therefore, classes that can be written in deriving are limited to those whose method types satisfy this restriction. However, note that this does not imply that references to the same type within AllTurtle's constructor can be used solely as read-only. For instance, a type possessing multiple references to the same type might be added as a constituent type, and we can imagine a method that swaps the contents of two of such references.

In general, this declaration of an assorted type:

```
data assorted T α1 ... αk = τ1 | ... | τm deriving (π1, ..., πn)
```

is translated as follows into a data type definition and instance declarations.

```
data T α1 ... αk = D1 τ1 | ... | Dm τm

instance Cast τ1 (T α1 ... αk) where
    ucast = D1
    dcast (D1 t) = Just t
    dcast _ = Nothing

⋮
```

And suppose, for example, that π_i in the deriving clause is of the form $C_i\ \beta_1\ \ldots\ \beta_j\ \_$. (The _ may appear in any position, but for convenience in the following explanation, it shall be considered to appear as the last argument.) Then, the following instance declarations are generated.

```
instance Ci β1 ... βj (T α1 ... αk) where
    method1 (D1 t) = fmap ucast (method1 t)
    ...
    method1 (Dm t) = fmap ucast (method1 t)
    method2 (D1 t) = fmap ucast (method2 t)
    ⋮
```

In other words, "_" is replaced with the assorted type being defined.

While fmap is used here for simplicity of explanation, whether it can be defined as an instance of Haskell's type class Functor is a separate matter, even if the argument type appears positively within the method's return value. For example, while an instance of Functor such as $(\mathsf{a} \to \mathsf{b}) \to (\mathsf{x}, \mathsf{a}) \to (\mathsf{x}, \mathsf{b})$ can be defined, an instance such as $(\mathsf{a} \to \mathsf{b}) \to (\mathsf{a}, \mathsf{y}) \to (\mathsf{b}, \mathsf{y})$ cannot. Therefore, in practice, it will be defined in a manner that involves inlining fmap.

3.4 Defaulting Plugin

To delay the definition of assorted algebraic data types as long as possible, we propose defining anonymous constructors and using overloading. So when exactly should the definition of an algebraic data type be fixed? It should be fixed when it is certain that no further constructors or methods will be added. Using methods such as ucast imposes constraints on type variables, such as Cast τ a, or type class constraints like $C_i \ldots$ a. However, since the type of the main function is ultimately (essentially) IO (), such type variables should disappear at some stage.

Even with the mechanism introduced at this point, one can explicitly provide a type using signature (::) immediately before the type variable disappears (is deemed ambiguous). While AllTurtle has a state type parameter s, making things slightly more complicated, one could write it as follows, for example:

```
do
  -- The definitions of x1, y1, t1, x2, y2, t2, and c2 are omitted.
  let turtle1 = Turtle x1 y1 t1
  let turtle2 = ColorTurtle x2 y2 t2 c2
  let turtles = (id :: [AllTurtle s] → [AllTurtle s])
                  [ucast turtle1, ucast turtle2]
  -- The code that uses turtles follows.
```

However, this is cumbersome, so we wish to enable automatic type resolution. When using variables such as turtles without explicitly specifying their type, an ambiguous type error should occur somewhere. Typically, ambiguous type errors occur with types of the form $\pi \Rightarrow \tau$, where a type variable appears in the type constraint π but not in the type body τ. (In practice, due to functional dependencies in type classes, ambiguity may not occur even when the variable does not appear in the body.) In such cases, no further constraints are added to the ambiguous type variable. When such ambiguity in a type variable is detected, we generate an assorted type from the type τ related to the type variable a via Cast τ a, and assign this assorted type to the ambiguous type variable.

In current Haskell, one can specify a default type when an ambiguous type error occurs. However, by default, defaults can only be specified for numeric types belonging to the Num class. Furthermore, the NamedDefault extension allows specifying default types for non-numeric types per class. The current NamedDefault extension cannot specify multi-parameter type classes, but the type classes targeted by this proposal should be treatable in the same manner, as they possess only one independent type parameter and the rest are dependent parameters (that is, they are so-called parametric type classes [3]). The ambiguity resolution mechanism proposed in this paper resembles the NamedDefault extension, but specifies default types for a collection of type classes rather than a single type class.

Given an assorted type definition such as the following,

```
data assorted T α1 ... αk = τ1 | ... | τm deriving (π1, ..., πn)
```

When the type constraints given for the ambiguous type variable γ are a subset of (Cast τ_1 γ, ..., Cast τ_m γ, π_1, ..., π_n) where _ is replaced with γ in π_i

and $\alpha_1, \ldots, \alpha_k$ are replaced with $\upsilon_1, \ldots, \upsilon_k$ respectively, then we can replace the type variable γ with $T\ \upsilon_1\ \ldots\ \upsilon_k$.

When multiple assorted types satisfy the condition, any one may be selected. The definition of an assorted type's method merely involves adding or removing tags $D_1, \ldots, D_m$; since the type variable is judged ambiguous, these tags never appear externally. Therefore, the result is the same regardless of which assorted type is chosen.

This assignment is implemented by GHC's compiler plugin (DefaultingPlugin). The defaulting plugin is invoked when ambiguous type variables are detected. It receives the type variable and information about the type constraint containing that variable, and is designed to return candidate assignments for the type variable. However, it became apparent that several issues exist when attempting to resolve ambiguity using this approach.

One issue arises from recursive assorted types. For instance, with a type such as AllTurtle, if a constraint like Cast (ContraryTurtle s α) β is applied, α and β should be unified since AllTurtle is defined as data AllTurtle s = ... | D3 (ContraryTurtle s (AllTurtle s)). However, when using the generic ucast, there is no necessity for this unification to occur before the plugin is invoked, so the type constraints concerning α and β are grouped separately. As the defaulting plugin is triggered for each ambiguous type variable, it cannot simultaneously handle type constraint information concerning both α and β.

Another concern is the potential for users to overuse functions such as ucast. Inserting more than two ucast calls where one would suffice (for example, before and after a function call) imposes constraints like Cast α γ and Cast γ β, increasing the effort required to eliminate the intermediate type variable γ. While this would not be impossible to implement the elimination, it increases the complexity of the mechanism.

Therefore, this proposal adopts the following approach. First, we prepare dedicated classes that provide casts from types that could potentially become components of the assorted type. For example, we define the following class for ContraryTurtle. (Currently, GHC's annotation (ANN) pragma is used to associate the type class FromContraryTurtle with the type ContraryTurtle.)

```
class FromContraryTurtle s e | e → s where
    fromContraryTurtle :: ContraryTurtle s e → e
{-# ANN type Turtle (CastFrom ''FromTurtle ''Turtle) #-}
```

For such annotated classes, instances are automatically generated when an assorted type containing ContraryTurtle is defined. For example, when AllTurtle is defined, instances like the following are automatically generated.

```
instance FromContraryTurtle s (AllTurtle s) where
    fromContraryTurtle = D3
-- FromTurtle, FromColorTurtle are automatically generated similarly.
```

The defaulting plugin will search for such classes instead of Cast. Then, this mechanism allows ambiguity resolution even in cases involving recursive assorted types, as demonstrated in the following example.

```
do
  -- The definitions of x1, y1, and t1 ··· are omitted.
  let turtle1 = Turtle x1 y1 t1
  let turtle2 = ColorTurtle x2 y2 t2 c2
  r2 ← newSTRef $ fromColorTurtle turtle2
  let turtle3 = ContraryTurtle r2
      turtles = [ fromTurtle turtle1,
                  fromColorTurtle turtle2,
                  fromContraryTurtle turtle3]
  mapM_ (λt → forward t 3.3) turtles···
```

4 Related Work

This section introduces related research on the expression problem that was not covered in Section 2.

4.1 Trees that Grow

Najd *et al.*'s "Trees that Grow" [10] proposes a technique for defining data types using type families, leaving room for later extension. For example, a data type is defined as follows:

```
data ExpX ξ = LitX (XLit ξ) Int
            | AddX (XAdd ξ) (ExpX ξ) (ExpX ξ)
            | ExpX (XExp ξ)
type family XLit ξ
type family XAdd ξ
type family XExp ξ
```

Then, by defining instances of these type families XLit, XAdd and XExp, we extend the data type.

This method allows us not only to increase the number of constructors but also to add fields to each constructor. However, it requires defining the data type in advance with the expectation of future extensions. Furthermore, repeated extensions can make accessing the extended parts inefficient. Without providing some form of syntactic sugar, defining instances of type families or pattern synonyms would become rather cumbersome.

4.2 Composable Data Types

Albers and Romeborn [1] propose an extension to Haskell's grammar for the expression problem. This proposal shares with ours the approach of extending syntax and converting it to standard Haskell. However, since its output code utilises Data Types à la Carte [14] and its extension, Compositional Data Types [2], it is considered to have low compatibility with reference types. Moreover, it requires extending a broad range of Haskell's grammar, encompassing not only type definitions but also function declarations.

5 Future Work

Several parts remain unimplemented due to unfinished design, or due to constraints in the compiler plugin specification. This section introduces such future challenges.

5.1 Binary Methods

Binary methods are methods that take two arguments of the same type, such as Eq or Ord. The proposed mechanism could potentially support binary methods, though several details require further refinement.

In general, for each component type constituting an assorted type, the argument types for a binary method must differ and should be represented using a multi-parameter type class as follows.

```
class Eq' a b where
    eq' :: a → b → Bool

class Ord' a b c | a b → c where
    compare' :: a → b → Ordering
    max, min :: a → b → c
```

The return type of a method may also depend on the argument type, as seen with c in max and min above. It could be either a or b, or it could be a type like Either a b.

In other words, generally binary methods are represented by multi-parameter type classes such as the following:

```
class BinaryMethod' α β γ | α β → γ where
    binaryMethod' :: α → β → γ
```

Then, the first and second arguments must be of different types, and the type appearing in the return value may vary for each combination, either per class or per method. The above γ may be α, may be β, or even may be `Either α β`.

To declare instances for such a class, the methods are defined for all the combinations of constituent types as follows.

```
instance BinaryMethod T where
    ⋮
    binaryMethod (D_i x) (D_j y) = fmap ucast (binaryMethod' x y)
    ⋮
```

A mechanism is needed to associate the auxiliary multi-parameter type classes Eq' and Ord' to the original binary method type classes Eq and Ord. Currently, we are considering using annotation pragmas to express this mapping and extending assorted types to type classes with binary methods.

5.2 GADT

Monads and abstract syntax trees also benefit from the ability to add new data types (constituents) later. These data types are generally defined as generalised algebraic data types (GADTs).

It seems possible to extend the syntax of the assorted type definition and the ambiguity resolution mechanism proposed in this paper to GADTs, but the current implementation does not support this. Enabling the definition of assorted types in GADTs and demonstrating useful application examples remains a future task.

In this paper, we have emphasised the usefulness of reference types for object-oriented programming. However, reference types seem also convenient in the implementation of embedded domain-specific languages (eDSLs). References may make it possible to employ features such as graph reduction and unification while utilising garbage collection of the host language. Therefore, demonstrating application examples in the implementation of eDSLs is also a future task.

5.3 Extensions requiring changes to the compiler plug-in

As mentioned in the section on defaulting plugins, the current (GHC 9.12.2) compiler plugin specification appears to trigger a plugin for each type variable judged ambiguous. However, when handling constraints for multi-parameter type classes like Cast, it becomes necessary to process information about multiple type variables simultaneously. Providing such an extension as a plugin would likely enable resolving ambiguities even when dealing with recursive types or multi-stage casts (e.g., casting to one assorted type, then casting again to another assorted type with more constructors but fewer corresponding methods).

6 Conclusion

We have explained that many existing proposals for the expression problem cannot be applied when data types include reference types, especially references to the same type as the object itself, and are therefore unsuitable for object-oriented programming techniques that require frequent use of references.

We have proposed a syntax for assorted types that can be applied to data types containing references, along with its translation and rules to resolve associated type ambiguities, and have implemented it as a compiler plugin. We also presented a program example demonstrating how type ambiguities can be resolved arising when creating heterogeneous containers for data types containing references to themselves.

The source code for implementing this proposal is available at https://github.com/KojiKagawa/hs-type-assort.git.

Acknowledgments. I would like to thank anonymous reviewers for their valuable comments. This study was supported by JSPS KAKENHI (grant number JP23K11350).

References

1. Albers, F., Romeborn, A.: A Syntax for Composable Data Types in Haskell, A User-friendly Syntax for Solving the Expression Problem. Master's thesis, Chalmers University of Technology and University of Gothenburg (2023)
2. Bahr, P., Hvitved, T.: Compositional data types. In: Proceedings of the seventh ACM SIGPLAN workshop on Generic programming. pp. 83–94. WGP '11, ACM, New York, NY, USA (September 2011). https://doi.org/10.1145/2036918.2036930
3. Chen, K., Hudak, P., Odersky, M.: Parametric type classes. In: Proceedings of the 1992 ACM Conference on LISP and Functional Programming. p. 170–181. LFP '92, Association for Computing Machinery, New York, NY, USA (1992), https://doi.org/10.1145/141471.141536
4. Garrigue, J.: Code reuse through polymorphic variants. In: Proceedings of the 2000 JSSST Workshop on Foundations of Software Engineering. pp. 93–100 (November 2000)
5. Kagawa, K.: Polymorphic variants in Haskell. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell. pp. 37–47 (2006). https://doi.org/10.1145/1159842.1159848
6. Kiselyov, O.: Typed tagless final interpreters. In: Generic and indexed programming: International spring school, sSGIP 2010, oxford, uK, march 22-26, 2010, revised lectures, pp. 130–174. Springer (2012)
7. Läufer, K.: Type classes with existential types. Journal of Functional Programming **6**(3), 485–518 (1996). https://doi.org/10.1017/S0956796800001817
8. Löh, A., Hinze, R.: Open data types and open functions. In: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming. pp. 133–144 (2006)
9. Marlow, S., et al.: Haskell 2010 language report. https://www.haskell.org/onlinereport/haskell2010/ (2010)
10. Najd, S., Peyton Jones, S.: Trees that grow. Journal of Universal Computer Science (JUCS) **23**, 47–62 (January 2017), https://www.microsoft.com/en-us/research/publication/trees-that-grow/
11. Odersky, M.: Objects and subtyping in a functional perspective. Tech. Rep. IBM Research Report RC 16423, IBM Research, Thomas J. Watson Research Center (January 1991)
12. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 71–84. Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/158511.158524
13. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 1–16 (2002)
14. Swierstra, W.: Data types à la carte. Journal of Functional Programming **18**(4), 423–436 (2008). https://doi.org/10.1017/S0956796808006758
15. Wadler, P.: The expression problem. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (November 1998)
16. Wright, A.K.: Simple imperative polymorphism. LISP and Symbolic Computation **8**(4), 343–355 (December 1995). https://doi.org/10.1007/BF01018828