Polymorphic Variants in Haskell

Koji Kagawa

RISE, Kagawa University
2217-20 Hayashi-cho, Takamatsu, Kagawa 761-0396, JAPAN
kagawa@eng.kagawa-u.ac.jp

Abstract

In languages that support polymorphic variants, a single variant value can be passed to many contexts that accept different sets of constructors. Polymorphic variants are potentially useful for application domains such as interpreters, graphical user interface (GUI) libraries and database interfaces, where the number of necessary constructors cannot be determined in advance.

The type system of Haskell, when extended with parametric type classes (or multi-parameter type classes with functional dependencies), has enough power to mimic polymorphic variants. This paper, first, explains how to encode polymorphic variants in Haskell's type system (Haskell 98 + popular extensions). However, this encoding of polymorphic variants are rarely used in practice. This is probably because it is quite tedious for programmers to write mimic codes by hand and because the problem of ambiguity would embarrass programmers.

Therefore, the paper proposes an extension of Haskell's type classes that supports polymorphic variants directly. This type system can produce vanilla Haskell codes as a result of type inference. Therefore it behaves as a preprocessor which translate the extended language into plain old Haskell. Programmers would be able to use polymorphic variants without worrying nasty problems such as ambiguities.

1. Introduction

1.1 Polymorphic Record and Variant Calculus

Extensions of the Hindley-Milner type system with polymorphic record and variant calculi have been extensively studied and known for years (*e.g.* [20, 22, 6]).

Variants and records are dual concepts in the theory of programming languages. Polymorphic record calculi allow a single function to be applied to many record types with different sets of labels. We can consider polymorphic record calculi as a basis of object-oriented programming languages. In this sense, polymorphic record calculi are widely used in the real world.

On the other hand, polymorphic variant calculi allow a single value to be passed to many functions which accept different sets of constructors. Theoretically, they are dual to polymorphic record calculi, and therefore we could use polymorphic variant calculi in order to introduce extensible algebraic datatypes into functional programming languages. And practically, there are some application domains where extensible algebraic datatypes are (potentially) useful, as we see in the next section.

1.2 Potential Applications of Polymorphic Variants

Suppose that we are writing an interpreter for a tiny language. We need a datatype for its abstract syntax.

```
data Expr = Var String | App Expr Expr
          | Lambda String Expr
```

Here, `Lambda "x" (Var "x")` is an internal representation of the expression "$\lambda x.x$." Then, for example, we can define the "eval" function for this datatype.

```
eval (Var x) env      = lookup x env
eval (App f e) env    = ...  (eval f env) ...
                        ...  (eval e env) ...
eval (Lambda x e) env = ...
```

Later, we may want a variation with a new constructor in order to treat specially, for example, full (saturated) function applications.

```
data ExprF extends Expr = FullApp ExprF [ExprF]
```

(This declaration means that `ExprF` is a datatype which has all the constructors of `Expr` as well as a new constructor `FullApp`. Note that this is a tentative syntax used for explanation only and not the one we will propose in this paper — we will introduce another declaration form later.) And then, we define, for example, the "print" function for this extended datatype.

```
print (Var str)      = show str
print (App e1 e2)    = ...  print e1 ...
                       ...  print e2 ...
print (FullApp f es) = ...
print (Lambda x e)   = ...  print e ...
```

On the other hand, we may want to keep `eval` being defined for only `Expr`, by converting `FullApp` into multiple `App`'s before values of the datatype are supplied into `eval`.

We can think of several variations and situations where we want to use slightly different datatypes for abstract syntax trees, which share, however, the core constructors.

1.3 A Problem of Polymorphic Variant Calculi

As you see, polymorphic variants are potentially useful for some domains. In practice, however, polymorphic variants are rarely used. The only practical programming language handling polymorphic variants is, to the author's best knowledge, Objective Caml since ver. 3 [4]. The Standard ML has *only one* extensible datatype `exn` — the type of exceptions. Haskell can mimic polymorphic variants using (multi-parameter) type classes. However, there are

some difficulties to be used in practice, which we will explain more precisely later (Section 2).

We could explain the reason of this situation rather abstractly using the following table, where Var. stands for polymorphic variant calculi and Rec. stands for polymorphic record calculi.

	existing functions	new functions
existing constructors	OK	Var.: OK Rec.: NG
new constructors	Var.: NG Rec.: OK	OK

In polymorphic record calculi, it is easy to add a new constructor — when a new constructor (or, in OO methodology, class) is added by extending an existing one with new fields, functions defined for the existing constructor can be still applied to objects of the extended type (left lower). On the other hand, in polymorphic variant calculi, though it is easy to add a new function defined by case analysis (right upper), when a variant type is extended by a new constructor, existing functions that use case analysis cannot be applied for the new constructor (left lower). Actually, the dual case of the latter situation corresponds to adding a new function to existing constructors in polymorphic record calculi (right upper). It is known to be hard to do this (to add a new method to existing classes) in object-oriented programming languages, and the "visitor pattern" [3] is invented for this very purpose.

The fact that it is difficult to add both new data constructors and new operations without modifying existing code is called *the extensibility problem* and has been extensively studied (*e.g.,* [2, 23, 17]). The problem, though symmetrical, seems to have severer impact on polymorphic variant calculi than on polymorphic record calculi, simply because functions are harder to define than constructors. For example, returning to `Expr` in the above example, we may want to add imperative features to the core language by extending the `Expr` datatype.

```
data ExprS extends Expr = Setq String ExprS
                        | Read | Write ExprS
```

Then, we want to somehow reuse for `ExprS` the `eval` function originally defined for `Expr`. We will explain the difficulty of this kind of reuse more in detail later (Section 2). Generally, for polymorphic variants to be useful in practical application domains, we would need some mechanisms to reuse existing functions for new constructors.

1.4 The Plan of the Paper

In this paper, we will show how polymorphic variants can be encoded in Haskell (Section 2). The encoding needs an extension of Haskell 98. The extension — multi-parameter type classes with functional dependencies [12] is, however, popular and available at least in two popular Haskell implementations Hugs and GHC. (We also require that *the monomorphism restriction* is turned off.) This encoding is not difficult to understand but at least tedious for programmers to write by hand, and is practially unfeasible. Therefore, we will propose a type system that directly supports polymorphic variants and records while avoiding the problem of ambiguity (Section 3). We will introduce a declaration form for polymorphic variants as a special form of (parametric) type class declarations. We also introduce a declaration form for polymorphic records and a new instance declaration form that treats records and variants symmetrically. We will explain these new forms by translating them into plain Haskell codes. Then, we will show some examples (Section **??**), discuss relations to existing work (Section 4), present future directions (Section 5) and summarize our contribution (Section 6).

2. Encoding Polymorphic Variants in Haskell

In this section, we will explain how to encode polymorphic variants in Haskell and how to reuse existing functions defined for polymorphic variants, solving the problem suggested in the Introduction. Then, we will make manifest the reason why the enconding is not used in practice, Still, the encoding itself explains the idea behind the new declaration forms that we will introduce in the next section.

2.1 Type Classes with Functional Dependencies

Haskell's type class system is a very general and powerful system for overloading. However, Haskell's type class system, as is defined in Haskell 98, cannot express polymorphic record and variant calculi, especially when we need parametric types such as *List* and *Tree*.

It is known, however, that the system of parametric type classes [1] — a generalization of Haskell's type class system — can encode polymorphic record and variant calculi. Type classes with functional dependencies [12] further generalize parametric type classes — parametric type classes are special cases where there is only one independent type parameter per class. Dependencies among type parameters are represented by a vertical bar in a class declaration:

```
class Foo a b c | a b → c where ...
```

Here, `a` and `b`, which appear on the left-hand side of →, are independent parameters, and `c`, which appears on the right-hand side of →, is a parameter dependent on `a` and `b`. This means that, if we have two predicates `Foo x y z` and `Foo x y w` that share the independent parameters `x` and `y` in a single predicate set, two dependent parameters `z` and `w` must be unified.

2.2 Encoding Polymorphic Variants

Polymorphic variants can be simply encoded as type classes where the sole independent parameter appears at the result type positions of member functions.

```
class List s x | s → x where
  cons :: x → s → s
  nil  :: s
```

Then, we can add constructors in subclasses:

```
class List s x ⇒ AppendList s x | s → x  where
  unit   :: x → s
  append :: s → s → s
```

We also have to define what we can call "standard instance types" of type classes, the types of whose constructors exactly match the class declarations

```
data T_List x =
  Cons_List x (T_List x) | Nil_List

data T_AppendList x =
    Cons_AppendList x (T_AppendList x)
  | Nil_AppendList
  | Unit_AppendList x
  | Append_AppendList (T_AppendList x)
                      (T_AppendList x)
```

If we can use GADT (Generalized Algebraic Data Types)-style declarations, correspondence betweeen class declarations and data type declarations will be much clearer.

```
data T_List :: * → *  where
  Cons_List :: x → T_List x → T_List x
```

```
  Nil_List  :: T_List x

data T_AppendList :: * → * where
  Cons_AppendList
    :: x → T_AppendList x → T_AppendList x
  Nil_AppendList    :: T_AppendList x
  Unit_AppendList   :: x → T_AppendList x
  Append_AppendList :: T_AppendList x
    → T_AppendList x → T_AppendList x
```

And of course, we need (rather trivial) instance declarations as well.

```
instance List (T_List x) x where
  cons = Cons_List
  nil  = Nil_List
instance List (T_AppendList x) x where
  cons = Cons_AppendList
  nil  = Nil_AppendList
instance AppendList (T_AppendList x) x where
  unit   = Unit_AppendList
  append = Append_AppendList
```

We can encode functions that accept polymorphic variants using constructors in standard instance types. For example,

```
lengthL Nil_List         = 0
lengthL (Cons_List _ xs) = 1 + lengthL xs

sumA Nil_AppendList = 0
sumA (Cons_AppendList x xs) = x + sumA xs
sumA (Unit_AppendList x)    = x
sumA (Append_AppendList xs ys) = sumA xs + sumA ys
```

Some functions (*e.g.* `sumA`) may have the case for `append` explicitly and other functions (*e.g.* `lengthL`) may be without the case for `append`. Then the constructors defined in `List` can be used for both kinds of functions as in `lengthL (cons 1 nil)` and `sumA (cons 2 nil)`. This is what polymorphic variant calculi exactly mean.

2.3 Reusing Functions

In order to reuse functions defined for `List` in its subclass, one may be temped to define a new function as follows:

```
lengthA (Append_AppendList xs ys) =
  lengthA xs + lengthA ys
lengthA (Unit_AppendList x) = 1
-- other constructors
lengthA (Cons_AppendList z zs) =
  lengthL (Cons_List z zs)
lengthA Nil_AppendList = lengthL Nil_List
```

Unfortunately, this does not type check since `zs` above may contain `append`'s as subcomponents of `zs` (*e.g.*, `cons 1 (append ... ... )`) and since `lengthL` is defined recursively.

Using a coercion such as:

```
coerce_AppendList_List
  :: T_AppendList x → T_List x
coerce_AppendList_List
  (Append_AppendList Nil_AppendList ys) =
  coerce_AppendList_List ys
coerce_AppendList_List (Append_AppendList xs ys) =
  Cons (hdA xs) (coerce_AppendList_List
                   (Append_AppendList (tlA xs) ys))

lengthA xs = lengthL (coerce_AppendList_List xs)
-- we omit definitions for these functions
hdA :: T_AppendList x → x
tlA :: T_AppendList x → x
```

works for `length`, however, is in general, not a good idea. It coerces deeply, that is, it entirely maps all the subcomponents of type `T_AppendList` to `T_List` and loses information. However, in general, a function may want the subcomponents of type `T_AppendList` to maintain their type. For example,

```
tlA xs = tlL (coerce_ApeendList_List xs)
```

is not a good definition since its type is

```
T_AppendList x → T_List x
```

instead of

```
T_AppendList x → T_AppendList x.
```

Thus, it is not easy to reuse functions for `List` in its subclasses. This seems to be the very reason why such extensible algebraic datatypes are not popular — it appears to be no use to extend the existing type, instead, we would rather rewrite the existing one as:

```
data List x = Nil | Cons x (List x)
            | Append (List x) (List x)
            | Unit x
```

and rewrite all the existing functions at the same time, losing much modularity. Functions that must be redefined may be scattered in the source program. Or even worse, no source file may be available when functions are defined in libraries.

2.4 Open Recursion

Objective Caml has polymorphic variants since version 3.0. As for the problem presented above, Garrigue [5] proposes using *open recursion*. It is not possible to completely rephrase O'Caml programs in Haskell. Therefore, we show his solution in pseudo Haskell codes, which are not actually typable in Haskell. (Of course, with simple modification, we can make it typable in Haskell.)

The idea is that we add an additional parameter to recursive functions that abstracts recursive invocation.

```
lengthL_aux le_rec Nil = 0
lengthL_aux le_rec (Cons _ xs) = le_rec xs
```

Here, the argument `le_rec` abstracts recursive invocation. Then, it is possible to reuse functions when a polymorphic variant is extended,

```
lengthA_aux le_rec (Append xs ys) =
    le_rec xs + le_rec ys
lengthA_aux le_rec (Unit x) = 1
lengthA_aux le_rec Nil = lengthL_aux le_rec Nil_
lengthA_aux le_rec (Cons x xs) =
    lengthL_aux le_rec (Cons x xs)
```

by simply "tying the knot" as follows.

```
lengthL = lengthL_aux lengthL
lengthA = lengthA_aux lengthA
```

Using this technique, we can reuse existing functions defined for less constructors.

2.5 Type Classes for Operations

However, in this technique, we must always provide a higher order function which abstracts recursive function invocation, whenever we define a recursive function for a variant which is to be extended. Fortunately, the system of type classes in Haskell can hide such higher order functions and administrative work from programmers.

Therefore, in Haskell, we can define `length` as a member function of a type class.

```
class Length a where
  length :: a → Int

instance Length (T_List x) where
  length Nil_List         = length_Nil
  length (Cons_List x xs) = length_Cons x xs

length_Nil = 0
length_Cons x xs = 1 + length xs

instance Length (T_AppendList x) where
  length Nil_AppendList         = length_Nil
  length (Cons_AppendList x xs) =
    length_Cons x xs
  length (Unit_AppendList x)    = length_Unit x
  length (Append_AppendList xs ys) =
    length_Append xs ys

length_Unit x = 1
length_Append xs ys = length xs + length ys
```

We do not have to write codes that explicitly take ane xtra argument.

However, still, a problem remains: if we write an expression such as:

```
length (cons (1::Int) (cons 2 nil))
```

the type checker reports that it has an ambiguous type.

```
(Length a, List a Int) => Int
```

That is, `cons (1::Int) (cons 2 nil)` has type `List a Int => a` and `length` has type `Length a => Int`. We cannot determine the type variable `a`.

For example, Hugs (started with command line option `-98`) reports the followng message.

```
ERROR "XX.hs":xx - Unresolved top-level overloading
*** Binding             : xxxx
*** Outstanding context : (Length b, List b Int)
```

In general, "Ambiguity" means that we have a type $\pi \Rightarrow \tau$ (π is a set of predicates and τ is a type in a narrow sense) where some free variables in π do not appear freely in τ (*i.e.* $FV(\pi) \not\subseteq FV(\tau)$ where FV stands for "free variables" as usual.) Then, programmers have to provide type annotations explicitly in order to disambiguate the meaning of the program. We can instantiate the type variable `a` to a concrete type, in this case, `T_List Int`. Therefore, we can insert a type annotation as follows:

```
length (cons 1 (cons 2 nil) :: T_List Int)
```

Or, when the type of the parameters is not completely known, we will have to write a little trickier code.

```
asList :: T_List x → T_List x
asList x = x

foo :: x → x → Int
foo x y = length (asList (cons x (cons y nil)))
```

2.6 Summary of Encoding

Now, we have finished an encoding of polymorphic variants in Haskell (+ type class with functional dependencies). It has some characteristics.

- We represent constructors of polymorphics variants as member functions of type classes.
- We also define functions that accepts polymorphic variants as member functions of type classes. This is necessary in order to make such functions reusable when variants are extended.

However, since the encoding uses type classes doubly — both for constructors and functions (operations), apparently, there are two major problems.

- It is tedious for programmers to write such mimic codes by hand.
- In general, it is not easy to add exact type annotations to all the necessary places in order to disambiguate type variables.

Haskell programmers instinctively avoid ambiguous types. This explains why this encoding mechanism has not been popular so far. However, in this case, ambiguity is not a sign of a pathological code. In fact, if we happen to instantiate the ambiguous type variable to another candidate type in the above example,

```
length (cons 1 (cons 2 nil) :: T_AppendList Int)
```

the meaning remains identical since the both codes use essentially the same branches for `length`.

Therefore, if we can leave the process of encoding to the compiler, we can use polymorphic variants more readily, which is the topic of the next section.

3. Variant and Record Declarations

We would like to design a type system and a set of declaration forms that directly supports polymorphic variants and has the same effect as the encoding explained in the previous section.

More specifically, in this section, we will introduce class declaration forms for polymorphic variants (constructors) as well as for methods (operations). (In order to guarantee that disambiguation of type variables does not affect meanings of programs, we must distinguish classes for methods from ordinary Haskell type classes.) We will also introduce instance relations between variants and methods. Then, we will explain how to translate these new declaration forms into plain Haskell code.

The new system has to do the following tasks:

- to define "standard instance types" for variants, and
- to declare instance relations between "standard instance types" and related classes.

Programmers do not have to know names of standard instance types and their constructors, they are "behind the scene" — they are all used completely internally by the compiler and programmers never use them explicitly in their programs.

The type system also has to do the following:

- to insert type annotations when ambiguous types concerning variant types appear.

Among them, the last one is non-trivial and we introduce a slight modification of the type inference algorithm for this purpose.

3.1 Variant Declarations

We introduce a new class declaration form in order to define polymorphic variants. The declaration form for polymorphic variants is almost the same as that of parametric type classes except that the keyword **variant** is used. (In the following, we use the syntax for parametric type classes: where we write the sole independent parameter on the left-hand side of the symbol $\in$, for we do not need the full power of type classes with functional dependencies and the notation of the former is a little more compact.)

variant $\overline{\pi} \Rightarrow \alpha \in \mathit{VariantName}\ \overline{\beta}$ **where**
  $\mathit{Constr}_1 :: \tau_1^1 \to \cdots \to \tau_1^{n_1} \to \alpha$

$\ldots$
$Constr_m :: \tau_m^1 \to \cdots \to \tau_m^{n_m} \to \alpha$

This introduces new symbols $Constr_1, \ldots, Constr_m$. (We will use identifiers beginning with capital letters for variants.) The restriction for **variant** declarations is:

- The independent variable α must appear as the type of the return value of functions. That is, functions must have types of the form $\cdots \to \alpha$. (It is possible to have a constructor with no argument — like `Nil` in the next example.)

The context $\overline{\pi}$ specifies superclasses as in **class** declarations in the current Haskell. Of course, super classes must be also variant classes. However, multiple inheritance is allowed.

Variant declarations are straightforwardly translated as type class declarations. That is, a declaration:

$$\texttt{variant}\ \alpha \in \mathit{VariantName}\ \overline{\beta}\ \texttt{where}\ \ldots$$

becomes a type class declaration:

$$\texttt{class}\ \mathit{VariantName}\ \alpha\ \overline{\beta} \mid \alpha \to \overline{\beta}\ \texttt{where}\ \ldots$$

where the type variable on the left-hand side of $\in$ becomes the sole independent parameter.

For example, the type of lists can be defined as:

```
variant xs ∈ List x where
  Nil  :: xs
  Cons :: x → xs → xs
```

The difference from ordinary **data** declarations is that we can add new constructors later:

```
variant xs ∈ List x ⇒ xs ∈ List2 x where
  Cons2 :: x → x → xs → xs

variant xs ∈ List x ⇒ xs ∈ AppendList x where
  Unit   :: x → xs
  Append :: xs → xs → xs
```

(Traditional **data** declarations can be considered as "final" **variant** declarations which cannot have subclasses.) In this example, we can think of `Cons2` as a "cdr-coded" list constructor (with only two elements, — of course, you can add as many elements as you wish). The variant declarations for `List` and `AppendList` are exactly translated into the type class declarations for the same names in the previous section.

Alternatively, it would be possible to adopt a syntax similar to **data** declarations.

```
variant List x = Nil | Cons x (List x)
```

However, recursive constructors such as `Cons` has arguments whose type contains the "self" type — the type of the return value. Their types must change when the variant type is extended. Here, we prefer to use a **class**-style syntax which makes this fact explicit by a type parameter (*i.e.* `xs`). (Another reason why we do not adopt a **data**-style syntax is a vague conjecture that variants will be further useful if they are combined with GADT (Generalized Algebraic Data Type)'s.)

Though we introduce variants as a special case of type classes, there is no need to declare datatypes (other than the *standard* instance types) as instances of variant classes.

- A variant class has no instances (other than its *standard* instance type and the *standard* instance types of its subclasses).

3.2 Record Declarations

We also introduce a new declaration form called record declarations in order to define functions (methods) that operate on variants. This is because if we did not treat record classes separately from ordinary Haskell classes, the ambiguity problem would remain — that is, the meaning of an expression would depend on the type which an ambiguous type variable is instantiated to. On the other hand, if we separate record classes from ordinary type classes, when an ambiguous type variable only concerns variant class and record class predicates, the meaning does not depend on the type it is instantiated to.

In "A Second Look at Overloading" [19], Odersky, Wadler and Wehr propose System O and solve the problem of ambiguity of type classes by putting a simple restriction on the types of symbols that can be overloaded. System O requires that overloaded symbols should be functions and that the type of the first argument should determine the actual implementation. (That is, overloaded functions must have type $\alpha \to \cdots$ where α is the placeholder variable of the class.) System O can encode polymorphic record calculi and more — it can, so to speak, add new "methods" or "fields" to existing datatypes.

We impose exactly the same restriction for method declarations in record classes. Moreover, in the sense that variant declarations have a symmetric restriction that overloaded symbols must have type $\cdots \to \alpha$ where α is the independent parameter, this paper proposes a system that can be considered as a symmetric extension of System O.

We use the keyword **record** instead of **class** to clarify that each overloaded operator obeys the System O restriction. We use the word **record**, because it can be seen as definition of selectors (methods) for records.

$\texttt{record}\ \overline{\pi} \Rightarrow \alpha \in \mathit{RecordName}\ \overline{\beta}\ \texttt{where}$
$\quad \mathit{method}_1 :: \alpha \to \tau_1$
$\quad \ldots$
$\quad \mathit{method}_m :: \alpha \to \tau_m$

This introduces new symbols $\mathit{method}_1, \ldots, \mathit{method}_m$. Here, α is the independent type variable and $\overline{\beta}$ is a sequence of type parameters dependent on α. The context $\overline{\pi}$ specifies superclasses in the same way as in **class** declarations in the current Haskell. Its meaning is the same as that of parametric type class declaration except for the restriction on the form of types:

- The independent variable α must appear as the type of the first argument of each function. (Functions must have types of the form $\alpha \to \cdots$.)

Therefore, a declaration:

$$\texttt{record}\ \alpha \in \mathit{RecordName}\ \overline{\beta}\ \texttt{where}\ \ldots$$

becomes a type class declaration:

$$\texttt{class}\ \mathit{RecordName}\ \alpha\ \overline{\beta} \mid \alpha \to \overline{\beta}\ \texttt{where}\ \ldots$$

where the type variable on the left-hand side of $\in$ becomes the sole independent parameter.

For example, a declaration:

```
record a ∈ Length where
   length :: a → Int
```

is exactly translated into the type class declaration for the same name in the previous section.

****

3.3 Instance Declarations

So far, we can consider **record** and **variant** declarations as special cases of **class** declarations in the traditional Haskell. Instance declarations are, however, different from the traditional ones.

In our system, variant classes cannot have instances in the usual sense. Instead, we declare a variant class (v) as an instance of a record class (r). It must have the form:

$$\textbf{instance}\ \overline{\pi} \Rightarrow v\ \overline{\tau} \ni \upsilon \in r\ \overline{\sigma}\ \textbf{where}$$
$$\quad \mathit{method}_m\ (\mathit{Constr}_n\ \ p_1\ \ldots\ \ p_{k_n}) = e_m$$
$$\quad \ldots$$

Here, υ between "$\ni$" and "$\in$" intuitively stands for the "self type" (the type of the first argument of method_m) and may appear in $\overline{\pi}$.

In order to keep the typing rule for instance declarations simple,

- we restrict constructors of polymorphic variants only appear as the toplevel constructor of the first argument in method definitions in instance declarations.

This is probably the simplest way to guarantee that methods in the record class accepts all the constructors in the variant class. If we allowed polymorphic variants to appear in other places in patterns, we would need more elaborate typing rules for patterns to guarantee that a certain method accepts all the variants in a certain set of variant classes. Moreover,

- we do not allow constructors of polymorphic variants appear outside of instance declarations.

Note that when v is a subclass of another class, we only provide cases for newly added constructors in v. Therefore, there is only one instance declaration for a specific $\mathit{method}/\mathit{Constr}$ pair.

```
instance List x ∋ xs ∈ Length where
   length Nil         = 0
   length (Cons x xs) = 1 + length xs

instance List2 x ∋ xs ∈ Length where
   length (Cons2 x y xs) =
     length (Cons x (Cons y xs))

instance AppendList x ∋ xs ∈ Length where
   length (Unit x) = 1
   length (Append xs ys) = length xs + length ys
```

Then, since the semantics of method_m does not depend on typing, no *ambiguity* will arise. In order for this to work, when we check the type of method_m in the above **instance** declaration, we must take into account that method_m may be added cases for new constructors in subclasses of v later and therefore, the current method definition for Constr_n might be later used with cases for new constructors. Then, we must make sure that the type of the method is not overly restricted so that it does not prevent later extensions. Therefore, the type checking rule for the instance declaration must ensure that:

- If the type of method_m and Constr_n are declared in variant and record declarations respectively as:

  $$\textbf{variant}\ \alpha \in v\ \overline{\beta}\ \textbf{where}$$
  $$\quad \mathit{Constr}_n :: \kappa_1 \to \cdots \to \kappa_{k_n} \to \alpha$$

  $$\textbf{record}\ \alpha \in r\ \overline{\gamma}\ \textbf{where}$$
  $$\quad \mathit{method}_m :: \alpha \to \mu_m$$

  the type of method_m in the above instance declaration should be as general as:

  $$\overline{\rho} \Rightarrow \upsilon \to \mu_m[\overline{\sigma}/\overline{\gamma}]$$

  where we assume the type of Constr_n to be

  $$\kappa_1[\overline{\tau}/\overline{\beta},\ \upsilon/\alpha] \to \cdots \to \kappa_{k_n}[\overline{\tau}/\overline{\beta},\ \upsilon/\alpha] \to \upsilon$$

  and $\overline{\rho}$ must be implied by $\overline{\pi} \cup \{\, \upsilon \in v\ \overline{\tau},\ \upsilon \in r\ \overline{\sigma}\,\}$ and υ stands for the "self type".

Moreover, the instance context $\overline{\pi}$ is subject to the same restriction as the traditional type classes: $\overline{\pi}$ must imply the contexts of all the superclass instances of r and v [21, page 47].

3.4 Type Inference

Basically, the core part of the type inference algorithm does not need to be changed and remains the same as the one described in [9]. We have to, however, change the behavior of "context reduction" since the form of instance relations has changed. Then, what should the type checker do for a type constraint set that contains both record and variant constraints?

Intuitively, it finds ambiguous types, checks whether we can instantiate the ambiguous type variable to a certain standard instance type, and then actually substitutes the ambiguous type variable to the standard instance type of the relevant variant classes.

The context reduction process of Haskell can be regarded as a special case of *simplification* in the terminology of Jones [10]. In our system, the corresponding process should be regarded as a combination of *simplification* and *improvement* since it involves a type substitution. We would formalize the process as a function named *impr*. It returns a pair of type substitution and a simplified type constraint set. Two auxiliary functions *check* and *find* are used in the definition of *impr*.

$$\begin{aligned}
\mathit{impr}(P) \quad = \quad & \textbf{let}\ V = \text{all variant class constraints in } P \\
& \qquad R = \text{all record class constraints in } P \\
& \qquad VR = \{\, (v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau}) \mid (\alpha \in v\,\overline{\sigma}) \in V, (\alpha \in r\,\overline{\tau}) \in R \,\} \\
& \textbf{in} \\
& \textbf{if}\ \forall\, (v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau}) \in VR.\ \mathit{check}(v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau}, P) \\
& \textbf{then}\ (\mathit{idSubst}, P) \\
& \textbf{else let}\ (v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau})\ \textbf{be}\ \text{an arbitrary pair} \\
& \qquad s.t. \neg \mathit{check}(v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau},\ P) \\
& \qquad (Q,\ v\,\overline{\gamma},\ \zeta,\ r\,\overline{\delta}) = \mathit{find}(v,\ r) \\
& \qquad S = \mathit{mgu}((\overline{\gamma},\ \zeta,\ \overline{\delta}), (\overline{\sigma},\ \alpha,\ \overline{\tau})) \\
& \qquad (S', P') = \mathit{impr}(S\ (P \cup Q))\ \textbf{in} \\
& (S' \circ S, P')
\end{aligned}$$

$$\begin{aligned}
\mathit{check}(v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau},\ P) \quad = \quad & \textbf{let}\ (Q,\ v\,\overline{\gamma},\ \zeta,\ r\,\overline{\delta}) = \mathit{find}(v,\ r)\ \textbf{in} \\
& \textbf{if}\ \text{there is a substitution } S \\
& \qquad s.t.\ S\ (\overline{\gamma}, \zeta, \overline{\delta}) = (\overline{\sigma}, \alpha, \overline{\tau})\ \textbf{and}\ S\ \pi \in P \\
& \textbf{then}\ \mathit{True}\ \textbf{else}\ \mathit{False}
\end{aligned}$$

$$\begin{aligned}
\mathit{find}(v,\ r) \quad = \quad & \textbf{if}\ \text{there is an instance declaration:} \\
& \qquad \textbf{instance}\ Q \Rightarrow v\,\overline{\gamma} \ni \zeta \in r\,\overline{\delta}\ \textbf{where}\ \ldots \\
& \textbf{then}\ (Q,\ v\,\overline{\gamma},\ \zeta,\ r\,\overline{\delta}) \\
& \textbf{else}\ \mathit{failure}
\end{aligned}$$

We assume that the standard improvement process for polymorphic type classes [10, § 3.1] (namely, if both $\alpha \in c\,\overline{\tau}$ and $\alpha \in c\,\overline{\sigma}$ are in P, the type parameters $\overline{\tau}$ and $\overline{\sigma}$ must be unified) is performed prior to our own simplification and improvement.

Note that, unlike context reduction in Haskell, the type checker usually do not discard any type constraints since other type constraints may be added later which may interact with them. The size of the type constraint set gets larger during recursive calls of *impr*. The *impr* function always terminates since it does not introduce new type expressions and the number of record and variant classes is limited.

The first element of the result of *impr* is a type substitution which is applied to the type and the type environment. The second element of the result replaces the type constraint set. Using the notation of [10], it is written as:

$$\frac{Q \mid TA \vdash^W E : \nu \qquad (T', P) = impr(Q)}{P \mid T'TA \vdash^W E : T'\nu}$$

It is not necessary to precisely specify *when* this simplification and improvement process is invoked. It must be done, however, at least before the type is presented to human and before the disambiguation process explained in the next paragraph takes place.

When α is an ambiguous type variable in $P \Rightarrow \tau$, we substitute α with the standard instance type of a set of the variant constraints given to α. Then, all the type constraints of the form "$\alpha \in \ldots$" can be safely discarded from P. At the same time, the type checker inserts a type annotation in the source program. (Ambiguous types can arise mainly after checking function applications.)

We have a prototype implementation of the proposed type inference algorithm by extending the engine of "Typing Haskell in Haskell" [11]. Most part of modification is necessary to incorporate functional dependencies. Therefore, the only essential enhancement in our implementation is the *impr* function presented above.

4. Related Work

We mentioned some related work already in the Introduction. In this section, we will refer to some others.

In [15], in order to construct modular interpreters, Liang, Hudak and Jones propose using a datatype `OR` that represents the disjoint union of two types, and a kind of subtyping relation:

```
data OR a b = L a | R b

class SubType sub sup where
  inj :: sub → sup
  prj :: sup → Maybe sub
```

An apparent drawback of their approach is inefficiency of data representation, since `OR` tend to be deeply nested. Here is an example taken from their paper.

```
type Term =
  OR (OR TermA (OR TermB (OR TermF
                          (OR TermL TermR))))
     (OR TermN (OR TermC (OR TermP TermT)))
```

More compact representation is desirable.

The type system of O'Haskell [18] has the notion of extensible datatypes. Unlike our system, it is based not on polymorphic record/variant calculi but on *subtyping*. Though superficially, the results look alike, their internal mechanisms are quite different. A drawback of such a subtyping approach is loss of information when we create heterogeneous collections.

Haskell++ [7] also supports a form of code reuse when we define a new datatype similar to an existing datatype. Without polymorphic variants, we have to represent heterogeneous lists using existential types [14], which also leads to loss of information.

The type system of Mondrian [16] allows both code reuse and heterogeneous lists in compensation for loss of some type safety property. This means that "message not understood" errors arise not in compile time but in run time.

HList [13] is a quite different approach to heterogeneous collections. Though it seems to concern much broader area than ours, as for heterogeneous collections, our approach seems more lightweight as we do not need type-level programming.

In "Extensible Algebraic Datatypes with Defaults," Zenger and Odersky [23] also tackle the problem of code reuse. Though it is presented as a new design pattern for Java, the underlying idea that newly added constructors can select a case branch for existing constructors seems to overlap with ours. In their system, the only possible existing case they can select is *the implicit default case*. Though it may be possible to extend their proposal to select cases other than the default case, it is likely that it would become too complex to write by hand.

Millstein, Bleckner and Chambers [17] also proposes a system in which both functions and datatypes are extensible. Though it can handle binary methods (such as *equality* and *set union*) more elegantly than ours, it does not deal with type inference.

5. Future Work

5.1 Binary Methods

In our system, it is possible to define binary methods, though it may look awkward. Binary methods are methods which take another argument of the same variant class.

```
record a ∈ Eq where
  (==), (/=) :: a → a → Bool
```

The syntactic restriction imposed on the instance declarations allows us to avoid the problem caused by "binary methods."

According to the limitation introduced in § 3.3, we do not allow an instance declaration such as:

```
instance a ∈ Eq ⇒ List a ∋ x ∈ Eq where
  Cons a as == Cons b bs = a==b && as==bs
  Nil       == Nil       = True
  Cons _ _  == Nil       = False
  Nil       == Cons _ _  = False
```

because it uses polymorphic variants in the pattern for the second parameter of the method. If we accepted this instance declaration, it would be possible to declare another instance such as:

```
variant a ∈ Foo where
   Foo :: Int → a

instance Foo ∋ x ∈ Eq where
  Foo n == Foo m = n == m
```

Nothing prevents a predicate set such as `(a ∈ Foo, a ∈ List Int, a ∈ Eq)` and as a result, an expression such as "`Foo 1 == Cons 1 Nil`" would become typable but would cause a runtime error.

Then, how should we define equalities for `List` (and `List2`)? We can circumvent this difficulty if we introduce an auxiliary class:

```
record x ∈ EqList a where
  eqCons :: x → (a, x) → Bool
  eqNil  :: x → Bool

instance (a ∈ Eq, x ∈ Eq) ⇒
        List a ∋ x ∈ EqList a where
  eqCons (Cons x xs) (y, ys) = x==y && xs==ys
  eqCons Nil         (_, _)  = False
  eqNil  (Cons _ _)          = False
  eqNil  Nil                 = True

instance x ∈ EqList a ⇒ List a ∋ x ∈ Eq where
  (Cons x xs) == ys = eqCons ys (x, xs)
  Nil         == ys = eqNil  ys
```

This is a well-known technique to deal with multimethods [8]. It becomes possible to declare subclasses of `List` as an instance of `EqList` later.

In practice, however, this is awkward and it would be necessary to automatically generate this kind of tedious instance declarations.

5.2 Default Definitions

In practice, many polymorphic variants would have similar behaviors for most method definitions. For example, as for `Cons2` and `Append`, we will define most methods as follows.

```
foo (Cons2 x y zs) = foo (Cons x (Cons y zs))
foo (Append xs ys) =
  foo (if isNull xs then ys
       else Cons (hd xs) (Append (tl xs) ys))
```

On the other hand, the current Haskell permits classes to have definitions of "default" methods.

```
class a ∈ Eq where
  (==), (/=) :: a → a → Bool
  x /= y = not (x == y)
```

Then, it would be possible to think of a similar mechanism for variants where the method definitions for a constructor C are automatically generated from a default definition.

We only have to write an empty instance declaration to get a method for `Cons2`. For example,

```
instance Cons2 x ∋ a ∈ Length where {}
```

Now, we can pass e :: `List2` x to, for example, `length` :: $xs \in$ `Length` $\Rightarrow xs \rightarrow$ `Int`. The constructor `Cons2` is converted to two `Cons`'s as is defined by the default declaration. Of course, if we do not declare `List2` as an instance of `Length`, the expression `length (Cons2 1 2 Nil)` is untypable.

We must investigate what kind of default mechanism is easy to use and intelligible to users. There are some questions to be answered. For example, if both the method and the variant involved in an instance declaration have default definitions, which has priority? What typing rule should apply to default definitions? For example, the default definition for `Append` has the constraint that three methods `isNull`, `hd` and `tl` must be defined for the type in question. What kind of type constraints should be allowed?

6. Conclusion

In this paper, we have explained how we can encode polymorphic variants in Haskell's type classes. Then, we have proposed a type system for polymorphic records and variants for Haskell. We have introduced:

1. a declaration form for polymorphic variants as a special case of parametric type classes,
2. a new instance declaration form between a "record" class and a "variant" class and rules corresponding to "context reduction" in the traditional Haskell, which can be explained as "simplification" and "improvement" in the terminology of Jones [10].

Moreover, the meanings of programs can be given independently of types and we need not worry about ambiguous type errors. Instead of avoiding ambiguous types altogether, the type system makes use of ambiguities even affirmatively.

The proposed type system can produce vanilla Haskell (Haskell 98 + type classes with functional dependencies) codes as a result of type inference. Therefore, the type system can behave as a preprocessor, and we can give the meanings of programs in the extended type system using translation to plain Haskell.

Acknowledgments

The author is grateful to Jacques Garrigue for valuable comments on previous drafts of this paper. Comments from anonymous referees on earlier versions of this paper were also helpful to simplify and to improve both the idea and the presentation.

An preliminary version of this paper is presented at APLAS'02 (The Third Asian Workshop on Programming Languages and Systems) whose proceedings are unpublished. The author is also grateful to the attendance of the workshop for helpful comments.

References

[1] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *ACM Conf. on LISP and Functional Programming*, June 1992.

[2] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, Sept. 1998.

[5] J. Garrigue. Code reuse through polymorphic variants. In *FOSE 2000*, Nov. 2000.

[6] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report Technical Report NOTTCS-TR-96-3, Computer Science, University of Nottingham, Nov. 1996.

[7] J. Hughes and J. Sparud. Haskell++: An object-oriented extension of Haskell. In *Haskell Workshop 1995*, 1995.

[8] D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA) 1986*, pages 347–349, 1986.

[9] M. P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992.

[10] M. P. Jones. Simplifying and improving qualified types. Reserch Report YALEU/DCS/RR-1040, Yale University, June 1994.

[11] M. P. Jones. Typing haskell in haskell. In *Proceedings of the 1999 Haskell Workshop*, pages 9–22, Oct. 1999.

[12] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, Mar. 2000. LNCS 1782.

[13] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proc. of the ACM SIGPLAN Haskell Workshop 2004*, pages 96–107, Sept. 2004.

[14] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[15] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, Jan. 1995.

[16] E. Meijer and K. Claessen. The design and implementation of Mondrian. In *Proceedings of Haskell Workshop 1997*, 1997.

[17] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *Proc. the 2002 ACM SIGPLAN International Conference on Functional Programming*, pages 110–122, Oct. 2002.

[18] J. Nordlander. Polymorphic subtyping in O'Haskell. In *Proc. the APPSEm Workshop on Subtyping and Dependent Types in Programming*, 2000. Ponte de Lima, Portugal.

[19] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–146, June 1995.

[20] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, Nov. 1995.

[21] S. Peyton Jones, J. Hughes, et al. *Haskell 98: A Non-strict, Purely Functional Language*, Feb. 1999. `http://www.haskell.org/onlinereport/`.

[22] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Annual ACM Symp. on Principles of Prog. Languages*, pages 77–88, January 1989.

[23] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the International Conference on Functional Programming*, September 2001.