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 can be used in order to introduce extensible algebraic datatypes into functional programming languages and 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. Very few functional languages, however, have a mechanism to extend existing datatypes by adding new constructors. In general, for polymorphic variants to be useful, we would need some mechanisms to reuse existing functions and extend them for new constructors.

Actually, the type system of Haskell, when extended with parametric type classes (or multi-parameter type classes with functional dependencies), has enough power not only to mimic polymorphic variants but also to extend existing functions for new constructors. This paper, first, explains how to do this in Haskell's type system (Haskell 98 with popular extensions). However, this encoding of polymorphic variants is difficult to use in practice. This is because it is quite tedious for programmers to write mimic codes by hand and because the problem of ambiguous overloading resolution would embarrass programmers. Therefore, the paper proposes an extension of Haskell's type classes that supports polymorphic variants directly. It has a novel form of instance declarations where records and variants are handled symmetrically.

This type system can produce vanilla Haskell codes as a result of type inference. Therefore it behaves as a preprocessor which translates the extended language into plain Haskell. Programmers would be able to use polymorphic variants without worrying nasty problems such as ambiguities.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Haskell, Type classes, Polymorphic variants, Extensibility

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'06 September 17, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-489-8/06/0009 … $5.00.

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.* [25, 30, 7]).

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. We can use polymorphic variant calculi in order to introduce extensible algebraic datatypes into functional programming languages. And there are some application domains where extensible algebraic datatypes are (potentially) useful, as we see in the next subsection.

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.

```
-- tentative syntax
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. We also conveniently assume that the recursive occurrences of `Expr` in the definition of `Expr` change to `ExprF` when we extend it to `ExprF`.) 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 passed into `eval` and may want expressions such as `eval (FullApp ... )` to be a type error.

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

Though theoretically, polymorphic variant calculi are dual to polymorphic record calculi and there are some application areas where extensible algebraic datatypes are potentially useful, polymorphic variants are rarely used in practice. The only practical programming language handling polymorphic variants is, to the author's best knowledge, Objective Caml since ver. 3 [5]. 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 that we will explain more precisely later (Section 2).

We can explain the reason of this situation somewhat abstractly using the following table.

		functions	
		existing	new
constructors	existing	OK	†
	new	‡	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 (‡ in the table above). On the other hand, in polymorphic variant calculi, though it is easy to add a new function defined by case analysis (†), when a variant type is extended by a new constructor, existing functions that use case analysis cannot be applied for the new constructor (‡). Actually, the dual case of the latter situation corresponds to adding a new function to existing constructors in polymorphic record calculi (†). It is known to be hard to do this (*i.e.* to add a new method to existing classes) in object-oriented programming languages, and the "visitor pattern" [4] 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 expression problem (aka the extensibility problem)* and has been extensively studied (*e.g.,* [3, 32, 22]). 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 example above, we may want to add imperative features to the core language by extending the `Expr` datatype.

```
-- tentative syntax
data ExprS extends Expr = Setq String ExprS
                        | Read | Write ExprS
```

Then, we cannot use the `eval` function directly but somehow want to reuse for `ExprS` the function that is 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 and extend existing functions for new constructors. Early attempts to introduce polymorphic records and variants to functional languages (*e.g.* [25, 30, 7]) simply treat polymorphic variants as a dual concept of polymorphic records and do not address this problem. Objective Caml has polymorphic variants since version 3.0 and Garrigue [6] proposes a solution that uses *open recursion* for this problem. This solution employs higher-order functions, which is, in some sense, reminiscent of implementation of type classes based on dictionary passing. Moreover, in OCaml, polymorphic records and polymorphic variants are separate datatypes. This is inconvenient since programmers have to decide in advance which they should use for each datatype in their programs.

1.4 Plan of the Paper

In this paper, we propose a type system for polymorphic variants and polymorphic records as an extension of Haskell's type classes with the following properties.

- It does not treat polymorphic variants and records separately but treats them as unified datatypes. At the same time, it solves the problem of reusing and extending functions presented above.
- If we only use symbols declared by our new declaration forms, the meanings of programs can be given independently of types.

From the second property, the proposed system can be considered as a symmetric extension of System-O [24] of Odersky, Wadler and Wehr.

The plan of this paper is as follow. First, in Section 2, we will show how we can encode polymorphic variants in Haskell (Section 2.2). The encoding needs an extension of Haskell 98. The extension – multi-parameter type classes with functional dependencies [14] is, however, available at least in two popular Haskell implementations Hugs and GHC and is explained beforehand in Section 2.1. (We also require that *the monomorphism restriction* is turned off.) We will explain the difficulty of reusing functions using concrete Haskell codes (Section 2.3) and present the existing approach for this problem used in Objective Caml (Section 2.4). We then present how this solution can be reformulated in Haskell (Section 2.5).

This encoding is not difficult to understand but at least tedious for programmers to write by hand, and is practically unfeasible. Therefore, we will propose a type system that directly supports polymorphic variants and records while avoiding the problem of ambiguous overloading resolution (Section 3). We will introduce a declaration form for polymorphic variants as a special form of (parametric) type class declarations (Section 3.1). We also introduce a declaration form for polymorphic records (Section 3.2) and a new instance declaration form that treats records and variants symmetrically (Section 3.3). We will explain these new forms by showing how to translate them into plain Haskell codes. Then, we will show some examples, (Section 4), discuss relations to existing work (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 partly the problem explained in Section 1. Then, we will show that problems still remain – we will make manifest the reason why the encoding is not used in practice. However, 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, as defined in Haskell 98, it 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 [2] – a generalization of Haskell's type class system – can

encode polymorphic record and variant calculi. Type classes with functional dependencies [14] further generalize parametric type classes – parametric type classes are special cases where there is only one independent type parameter per class. Declarations of dependencies among type parameters are written after 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 equal.

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. Here, we switch to another simple example of polymorphic variants – lists with some non-standard constructors.

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

Later, 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 some associated datatypes.

```
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)
```

We call T_List the "standard instance types" of type class List. The types of the constructors of T_List correspond to the types of the methods in the class declaration.

If we can use GADT (Generalized Algebraic DataType)-style [29] declarations, the correspondence of class declarations with datatype declarations is much clearer. (In the declaration of a GADT, we can give the type signatures of constructors explicitly after the keyword **where**.)

```
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 cannot use member functions such as cons and nil in patterns. Instead, we can encode functions that accept polymorphic variants using constructors of standard instance types. For example,

```
hdL (Cons_List x _)      = x

tlL (Cons_List _ xs)     = xs

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 member functions defined in List can be used for both functions as in `lengthL (cons 1 nil)` and `sumA (cons 2 nil)`, where they accept different sets of constructors. 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 (*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 → T_AppendList 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 much information. However, in general, a function may want the subcomponents of type T_AppendList to maintain their type. For example,

```
tlA xs = tlL (coerce_AppendList_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 [6] proposes using *open recursion*. We show his solution in Haskell code.

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. (Careful readers may have noticed that the code above give lengthL_aux and lengthA_aux (and therefore lengthL and lengthA) the same type and also that simply applying the "standard instance types" technique presented in Section 2.2:

```
lengthL_aux le_rec Nil_List = 0
lengthL_aux le_rec (Cons_List _ xs) = le_rec xs

-- lengthA_aux does not type check.
lengthA_aux le_rec (Append_AppendList xs ys) =
    le_rec xs + le_rec ys
lengthA_aux le_rec (Unit_AppendList x) = 1
lengthA_aux le_rec Nil_AppendList =
    lengthL_aux le_rec Nil_List
lengthA_aux le_rec (Cons_AppendList x xs) =
    lengthL_aux le_rec (Cons_AppendList x xs)
```

does not make the code typeable in Haskell. On the other hand, OCaml's type system can treat recursive constructors such as Cons flexibly and can give lengthL and lengthA different types. This flaw, however, does not appear in the solution based on type classes explained next.)

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 :: Int
length_Nil = 0

length_Cons :: Length xs ⇒ x → xs → Int
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 → Int
length_Unit x = 1

length_Append :: Length xs ⇒ xs → xs → Int
length_Append xs ys = length xs + length ys
```

We do not have to write codes that explicitly take an extra parameter.

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

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

the type checker reports that it has an ambiguous type.

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

That is, cons 1 (cons 2 nil) has type List a Int => a and length has type Length a => a → Int. We cannot determine the type variable a. (To keep matters simple, we assume that literals such as 1 and 2 have type 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.") 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 (or have to use scoped type variables [28]).

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

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

-- or, using a scoped type variable
-- foo x y =
-- let tmp :: T_List x = cons x (cons y nil)
-- in length tmp
```

2.6 Summary of Encoding

Now we have presented an encoding of polymorphic variants in Haskell (using type class with functional dependencies). There are some points worth noticing:

- We represent constructors of polymorphic variants as member functions of type classes.
- We also represent operations that accept 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), there are some apparent 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 unresolved 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 `T_AppendList Int` 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 translation 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.

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 codes.

The new system has to do the following tasks:

- to generate the definition of "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 generated standard instance types and their constructors – 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, 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.)

$$\begin{array}{l}\textbf{variant}\ \overline{\pi} \Rightarrow \alpha \in \mathit{VariantName}\ \overline{\beta}\ \textbf{where} \\ \quad \mathit{constr}_1 :: \tau_1^1 \to \cdots \to \tau_1^{n_1} \to \alpha \\ \quad \ldots \\ \quad \mathit{constr}_m :: \tau_m^1 \to \cdots \to \tau_m^{n_m} \to \alpha\end{array}$$

This introduces new symbols $\mathit{constr}_1, \ldots, \mathit{constr}_m$. (Remember that we write the sole independent parameter on the left-hand side of the symbol $\in$ in the notation of parametric type classes. Therefore,

```
class a ∈ Foo b c where ...
```

can be rewritten as

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

using the notation of type classes with functional dependencies.) Though we refer to these symbols as "variant constructors" or simply "constructors," we allow them to appear in patterns in only limited places. We will return to this topic when we introduce a new form of instance declarations in Section 3.3.

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 parameter – like `nil` in the next example.)

The context $\overline{\pi}$ specifies superclasses as in **`class`** declarations in the current Haskell. Super classes must be also variant classes.

We also require that the restriction of the form of the context is exactly as is proposed by Peyton Jones, Jones and Meijer [27, § 4.1 and § 4.8]: that is, constraints in the super class context must be of the form $D\ \gamma_1\ \ldots\ \gamma_n$ where the γ_i are distinct, and are in a subset of α and $\overline{\beta}$. This is necessary in order to guarantee termination of context reduction, as we will see shortly. There are no other restrictions on superclass context. Especially, multiple inheritance is allowed.

Variant declarations are straightforwardly translated into plain type class declarations. That is, a declaration of the form:

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

is translated into a type class declaration:

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

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 Section 2.2.

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

```
-- alternative syntax
variant List x = Nil | Cons x (List x)
```

However, recursive constructors such as `Cons` has parameters 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 that makes this fact explicit by a type parameter (*i.e.* `xs`). Moreover, as we conjecture that polymorphic variants will be further useful if they are combined with GADTs (Generalized Algebraic Data Types) [29], we use a syntax that can be extended to GADTs.

Though we introduce variants as a special case of type classes, Programmers cannot declare datatypes as instances of variant classes.

- A variant class cannot have user-defined instances (except that its standard instance type and the standard instance types of its subclasses become automatically its instances).

3.2 Record Declarations

We also introduce a new declaration form called record declarations in order to define functions (methods) that operate on variants. 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. Since 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" [24], 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.

```
record π̄ ⇒ α ∈ RecordName β̄ where
  method_1 :: α → τ_1
  ...
  method_m :: α → τ_m
```

This introduces new symbols $method_1, \ldots, method_m$. 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 $\alpha \to \cdots$. Also α can appear in type of the second, third and other arguments as well as the return type.)

We also require that the restriction of the form of the context is exactly as is proposed by Peyton Jones, Jones and Meijer [27, § 4.1 and § 4.8] in the same way as variant declarations.

Therefore, a declaration of the form:

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

is translated into a type class declaration:

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

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

For example, a record declaration:

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

is exactly translated into the type class declaration for the same name in Section 2.5.

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, quite 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:

```
instance π̄ ⇒ V τ̄ ∋ υ ∈ R σ̄ where
  method_m (constr_n p_1 ... p_{k_n}) = e_m
  ...
```

Here, υ between "$\ni$" and "$\in$" intuitively stands for the "self type" (the type of the first parameter of $method_m$) and may appear in $\overline{\pi}$. (Though we use the same keyword **instance** as ordinary instance declarations, they are syntactically distinguishable.)

The most notable restriction of our system lies in places where we allow constructors of polymorphic variants (symbols introduced by our **variant** declarations) to appear in patterns.

- we allow constructors of polymorphic variants to only appear as the *toplevel* constructor of the *first* parameter in method definitions in instance declarations.
- we do *not* allow constructors of polymorphic variants to appear outside of instance declarations.

This means we do not allow variant constructors to appear in $p_1 \ldots p_{k_n}$ nor patterns in e_m in the declaration above (nor outside of the instance declaration). Though this restriction may seem overly severe, we will be discussing this restriction later again in Section 4.

This is 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 to give elaborate typing rules for patterns to guarantee that a certain method accepts all the variants in a certain set of variant classes.

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 *method*/*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 we may add to $method_m$ additional instances for new constructors in subclasses of V later and therefore, we may use the current method definition for $constr_n$ later with definitions for new other 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 of the form:

```
instance π̄ ⇒ V τ̄ ∋ υ ∈ R σ̄ where
  method_m (constr_n p_1 ... p_{k_n}) = e_m
  ...
```

must ensure that:

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

  variant $\alpha \in V\ \overline{\beta}$ **where**
  $constr_n :: \kappa_1 \to \cdots \to \kappa_{k_n} \to \alpha$

  record $\alpha \in R\ \overline{\gamma}$ **where**
  $method_m :: \alpha \to \mu_m$

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

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

  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". (The notation $\tau[\overline{\sigma}/\overline{\alpha}]$ stands for a type where type variables $\overline{\alpha}$ in τ are substituted by types $\overline{\sigma}$ respectively.)

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 [26, page 47]. (System-O has a restriction that amounts to saying $FV(\overline{\tau}) \subseteq FV(\overline{\sigma})$ in the notation of our system in order to guarantee termination of unification. This restriction, however, does not seem necessary in our system.)

This new form of instance declarations is translated into definitions of explicitly typed functions:

$$\begin{aligned} method_m\_constr_r &:: \overline{\pi} \cup \{\, \upsilon \in V\ \overline{\tau},\ \upsilon \in R\ \overline{\sigma}\,\} \Rightarrow \\ &\quad \kappa_1[\overline{\tau}/\overline{\beta},\ \upsilon/\alpha] \to \cdots \to \kappa_{k_n}[\overline{\tau}/\overline{\beta},\ \upsilon/\alpha] \\ &\quad \to \mu_m[\overline{\sigma}/\overline{\gamma},\ \upsilon/\alpha] \\ method_m\_constr_r\ &p_1 \ldots\ p_{k_n} = e_m \end{aligned}$$

where the identifier $method_m\_constr_r$ is put into a name space that is different from that of user defined functions. Later, we use these functions when we need to generate instance declarations for standard instance types. Therefore, separate compilation poses no problems.

For example, the instance declarations for `List`, `AppendList` and `Length` above are translated into functions `length_Nil`, `length_Cons`, `length_Unit` and `length_Append` given in Section 2.5.

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 by Jones [11]. 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 involved variant classes.

The context reduction process of Haskell can be regarded as a special case of *simplification* in the terminology of Jones [12]. 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*.

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

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

$find(V,\ R)$ = **if** there is an instance declaration:

instance $Q \Rightarrow V\ \overline{\gamma}\ \ni\ \zeta\ \in\ R\ \overline{\delta}$ **where** ...

then $(Q,\ V\ \overline{\gamma},\ \zeta,\ R\ \overline{\delta})$

else *failure*

We assume that the standard improvement process for multiple-parameter type classes with functional dependencies [12, § 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 plain Haskell, the type checker does not discard any type constraints since other type constraints may be added later which may interact with them. Therefore, 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 due to restrictions in the form of super class constraints and the number of record and variant classes is finite. (The size of the constraint set cannot increase forever.)

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 Jones [12], it is written as:

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

This simplification and improvement process must be invoked at least before the disambiguation process takes place (and before the type is presented to human). Since ambiguous types can arise after type-checking function applications, it must be invoked after checking function applications.

For translating the code into plain Haskell, when the type checker find an ambiguous type variable α in a type $P \Rightarrow \tau$, it inserts an explicit type annotation in the source program in order to disambiguate the type variable α. Then, all the type constraints of the form "$\alpha \in \ldots$" can be safely discarded from P. And in order for the type annotation to make sense, it must do following things.

- It defines the "standard instance type" of a set of the variant constraints given to α (if it is not yet defined). That is, let $\{\ \alpha \in V_i\ \overline{\sigma_{i,j}}\ \}$ be the set of variant constraints given to α in P and let $constr_{i,j} :: \tau^1_{i,j} \rightarrow \cdots \rightarrow \tau^{n_{i,j}}_{i,j} \rightarrow \alpha_i$ be the constructors defined for V_i, we define a new data type

  ```
  data Foo β̄ α where
    Constr_{i,j}_Foo :: τ^1_{i,_j}[(Foo β̄ α)/α_i] → ··· →
                        τ^{n_{i,j}}_{i,_j}[(Foo β̄ α)/α_i] → Foo β̄ α
  ```

  (Here, we use the GADT-style syntax for convenience. Here, $\overline{\beta}$ is a sequence of type parameters that appear in the type of $constr_{i,j}$.)

- It gives instance declarations that the standard instance type given above is an instance of variant classes $\{\ V_i\ \}$.

  ```
  instance V_i (Foo β̄ α) γ̄ where
    constr_{i,j} = Constr_{i,j}_Foo
  ```

  $\overline{\gamma}$ is a subsequence of $\overline{\beta}$ that is related to V_i.

- It gives instance declarations that the standard instance type above is an instance of record classes $\{R_m\}$ where $\{\alpha \in R_m\ \overline{\sigma_{m,n}}\}$ is a set of record constraints given to α in P.

  ```
  instance R_m (Foo β̄ α) γ̄ where
    method_p (Constr_{i,j}_Foo x̄) = method_p_constr_{i,j} x̄
  ```

  The $method_p\_constr_{i,j}$ is a set of explicitly typed functions generated from (variant × record)-**instance** declarations (*e.g.* `length_Nil` and `length_Cons`).

For example,

```
length (cons 1 (cons 2 Nil))
```

has an ambiguous type

```
(α ∈ List Int, α ∈ Length) ⇒ Int
```

and therefore is translated into the following code.

```
length (asList (cons 1 (cons 2 (Nil))))
```

where the definition of `asList`, a related datatype `T_List` and instance declarations "**instance** `List (T_List x) x` ... " and "**instance** `Length (T_List x)` ... " are exactly the same as those presented in Section 2.5.

We have a prototype implementation of the proposed type inference algorithm [15] by extending the engine of "Typing Haskell in Haskell" [13]. Most part of extension is for incorporating functional dependencies. Therefore, the only essential enhancement in our implementation is the *impr* function presented above.

4. Examples and Discussion

In this section, we will present some examples. At the same time, we will discuss some design decisions of our system.

4.1 Acceptor Classes

In our system, we can use polymorphic variant constructors at only limited places in instance declarations. This is certainly a severe restriction, since we must always define record classes in order to define operations that accept polymorphic variants and we cannot use `case` expressions for polymorphic variants. This restriction, however, can be, to some extent, compensated by defining "acceptor classes" as a special case of record classes.

For polymorphic record calculi, System-O considers typing rules for record literals $\{l_1 = e_1, \ldots, l_n = e_n\}$. It proposes an encoding where we define a fresh datatype for every set of record labels.

```
data R_{l_1 ... l_n} a_1 ··· a_n = R_{l_1 ... l_n} a_1 ··· a_n

instance R_{l_1 ... l_n} a_1 ··· a_n ∈ Class_{l_i} a_i where
  l_i = λ (R_{l_1 ... l_n} x_1 ··· x_n) → x_i
```

We need to give a similar encoding for polymorphic variants.

As an example, let us take `sum`, which we may want to define as follows.

```
-- not allowed in our system
sum (cons x xs) = x + sum xs
sum nil         = 0
```

Of course, this definition is not possible in our system, when `sum` is not a member function of a record class. In order to avoid defining record classes for each list operations, we define a new class (say, `ListAcceptor`) where `xs ∈ ListAcceptor x` intuitively means that `xs` has *at most* two constructors `cons` and `nil`, while `xs ∈ List x` intuitively means `xs` has *at least* constructors defined in the variant class `List x`. (This "acceptor" constraint seems to correspond to an upper bound "`xs[< Cons x xs | Nil]`" of Objective Caml ver. 3 [5].)

```
record xs ∈ ListAcceptor x where
  toListView :: xs → ListView x xs
```

where `ListView` is a datatype defined as:

```
data ListView x xs = ConsView x xs | NilView
```

And we rewrite sum as follows.

```
sum = sumView ∘ toListView
```

where

```
sumView (ConsView x xs) = x + sum xs
sumView NilView         = 0
```

Then, the type of sum becomes:

```
sum :: xs ∈ ListAcceptor Int ⇒ xs → Int
```

Every expression that wants to use case analysis for List must be rewritten in such a way that constructors are converted into ListView constructors, composed with "toListView" function and will be given ListAcceptor constraint.

Of course, List x is a trivial instance of ListAcceptor x.

```
instance List x ∋ xs ∈ ListAcceptor x where
  toListView (cons x xs) = ConsView x xs
  toListView nil         = NilView
```

Note that this toListView function does not coerce the argument deeply as coerce_AppendList_List does in Section 2.3. Then, we can write expressions such as follows.

```
sum (cons 1 (cons 2 nil))
```

Similarly, every variant class can have its corresponding "view" datatype (ListView in the above example) where the "self" type is replaced by an extra type parameter (xs in the definition of ListView above) and the "acceptor" class is a class which has only one method that trivially translates the argument to an object of the "view" type.

It might be possible to automatically generate this kind of "acceptor" classes for every variant class. (Otherwise, it would be better to define sum in a record declaration and make it extensible.) In this paper, however, we refrain from doing this. Unlike "standard instance types" that are unseen from programmers, it would involve classes (···Acceptor) and types (···View) that might appear as output of type inference and observable from programmers.

Interestingly, we can declare instance relations also between subclasses of List such as List2 and AppendList, and the ListAcceptor class.

```
instance List2 x ∋ xs ∈ ListAcceptor x where
  toListView (cons2 x1 x2 xs)
    = toListView (cons x1 (cons x2 xs))

isNull :: xs ∈ ListAcceptor x ⇒ xs → Bool
isNull = isNullView ∘ toListView

isNullView (ConsView _ _) = False
isNullView NilView        = True

hd :: xs ∈ ListAcceptor x ⇒ xs → x
hd = hdView ∘ toListView

hdView (ConsView x _)     =  x

tl :: xs ∈ ListAcceptor x ⇒ xs → xs
tl = tlView ∘ toListView

tlView (ConsView _ xs)    = xs

instance AppendList x ∋ xs ∈ ListAcceptor x where
  toListView (unit x) = toListView (cons x nil)
  toListView (append xs ys)
    = toListView (if isNull xs then ys
                  else cons (hd xs)
                            (append (tl xs) ys))
```

Then, it is possible to pass a value that consists of unit and append to operations such as sum that are defined for only ConsView and NilView as follows.

```
sum (append (unit 1) (unit 2))
```

In this example, polymorphic variants behave like views [31, 1]: we represent lists using a set of constructors unit and append, while we still define functions for lists as if lists are constructed only from cons and nil.

4.2 Binary Methods

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

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

The syntactic restriction imposed on the instance declarations allows us to circumvent problems typically caused by binary methods.

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

```
-- not allowed
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 (as are underlined). 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
```

Our type system would not prevent 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 typeable but would cause a runtime error.

Then, how should we define equalities for List? 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 [10]. It becomes possible to declare subclasses of List as an instance of EqList later. Since the pair of constraints (a ∈ List Int, a ∈ Eq) requires an additional constraint "a ∈ EqList Int" and the

relation "`Foo ∋ a ∈ EqList Int`" does not hold, the expression "`foo 1 == cons 1 nil`" does not type check.

In practice, however, it might be necessary to automatically generate this kind of tedious instance declarations, possibly by allowing polymorphic variant constructors to appear in the second (third, ...) parameter of methods. It would be also possible to support nested patterns (such as `cons x nil`) by a similar technique (using auxiliary acceptor classes).

5. Related Work

We mentioned some related work already in the Introduction (Section 1). In this section, we will refer to some others.

In order to construct modular interpreters, Liang, Hudak and Jones [18] 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 [23] 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++ [9] 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 [17], which also leads to loss of information.

The type system of Mondrian [21] 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 [16] is a quite different approach to heterogeneous collections. It is more ambitious and seems to concern much broader area than ours. For example, heterogeneous list constructors are defined by utilizing dependent type programming in Haskell [8, 20].

```
data HNil      = HNil
data HCons e l = HCons e l

class HList l
instance HList HNil
instance HList l ⇒ HList (HCons e l)
```

Operations on list such as `head`, `tail` and `append` are also formulated at the type level.

```
class HAppend l l' l'' | l l' → l'' where
  hAppend :: l → l' → l''

instance HList l ⇒ HAppend HNil l l where
  hAppend :: HNil = id

instance (HList l, HAppend l l' l'')
    ⇒ HAppend (HCons x l ) l' (HCons x l'') where
    hAppend (HCons x l) = HCons x ∘ hAppend l
```

Collections tend to have much more verbose types in HList than in our system. When we define heterogeneous container types other than the list type, we need analogous definitions at the type level. In their paper, heterogeneous lists thus defined are used for representing extensible records. The term "extensible" here means an ability to add new labels into existing *values* of record types. It is different from extensibility of datatypes and functions when we speak of the expression problem. It seems that the expression problem itself is not directly addressed in their paper.

Millstein, Bleckner and Chambers [22] also propose a system in which both functions and datatypes are extensible. It can handle binary methods such as *equality* and *set union*. In order to ensure modular typechecking, it uses a kind of subtyping and requires that some functions have global default cases. This is not fully satisfactory, since from the result of type checking, one cannot be sure whether the default case is used or a specific case is used for a particular constructor.

Löh and Hinze [19] propose a system that supports both open (extensible) datatypes and open functions for Haskell. Unlike ours, it does not impose restrictions as to where constructors of open datatypes can appear in patterns. In order to interpret such patterns, it collects all left hand sides of an open function into *Main* module, and employs *best-fit pattern matching* instead of *first-fit pattern matching*. In their system, if we extend a datatype with a new constructor but do not extend a function with a new case and call the function on arguments that contains a new constructor, we do not get a type error but get a pattern match failure at runtime. This is unsatisfactory – in object-oriented languages, objects of class *Foo* and objects of class *Bar* (where *Bar* is a subclass of *Foo*) can coexist and can be properly typechecked, avoiding "message not understood" errors at run-time.

6. Conclusions

In this paper, first, 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 well as polymorphic records, 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 [12].

We can extend datatypes by adding new constructors and can extend functions that accept them as well. Zenger and Odersky [33] propose five requirements that a solution to the expression problem should satisfy – (1) extensibility in both dimensions, (2) strong static type safety, (3) no modification or duplication, (4) separate compilation, (5) independent extensibility. The system proposed in this paper satisfies all five requirements.

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 (*i.e.* Haskell 98 using 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. The prototype is available from: `http://guppy.eng.kagawa-u.ac.jp/~kagawa/PVH`.

Acknowledgments

The author is grateful to Jacques Garrigue for valuable comments on previous drafts of this paper. Comments from Haskell Workshop 2006 referees and comments from referees on earlier versions of this paper were 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] W. Burton, E. Meijer, P. Sansom, S. Thompson, and P. Wadler. Views: An extension to Haskell pattern matching, Oct. 1996. `http://www.haskell.org/development/view.html`.

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

[3] 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.

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

[5] J. Garrigue. Programming with polymorphic variants. In *The 1998 ACM SIGPLAN Workshop on ML*, Sept. 1998.

[6] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering (FOSE) 2000*, Nov. 2000.

[7] 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.

[8] T. Hallgren. Fun with functional dependencies. In *Proceedings of the Joint CS/CE Winter Meeting*, pages 135–145, Jan. 2001. `http://www.cs.chalmers.se/~hallgren/Papers/wm01.html`.

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

[10] 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.

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

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

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

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

[15] K. Kagawa. Polymorphic variants in Haskell (prototype implementation), 2006. available from `http://guppy.eng.kagawa-u.ac.jp/~kagawa/PVH`.

[16] 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.

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

[18] 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.

[19] A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN symposium on Principle and Practice of Declarative Programming (PPDP 2006)*, July 2006.

[20] C. McBride. Faking it (simulating dependent types in Haskell). *Journal of Functional Programming*, 12(4&5):375–392, 2002. Special Issue on Haskell.

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

[22] 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.

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

[24] 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.

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

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

[27] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: exploring the design space. In *Haskell Workshop 1997*, 1997.

[28] S. Peyton Jones and M. Shields. Lexically scoped type variables, 2004.

[29] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*, 2006.

[30] 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.

[31] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, 1987.

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

[33] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *Proceedings of the 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*, Jan. 2005.