# Bidirectional Type Class Instances

Koen Pauwels
koen.pauwels@student.kuleuven.be

Katholieke Universiteit Leuven, Leuven, Belgium

September 3, 2018

**Abstract**

In this work, we show that the interaction between Generalized Algebraic Datatypes and type classes can lead to unpleasant surprises in Haskell. We show that this issue is caused by the current unidirectional interpretation of type class instance declarations, which means the compiler cannot make certain deductions that seem obvious to the programmer. We propose an extension to the type system that lets the compiler treat all instance declarations as bidirectional.

**Keywords:** Haskell, Type classes

**Motivation**

Consider the following *Term* GADT, which encodes a simple expression language with a single "pair" operator, similar to the example given by Johann and Ghani [1]:

```
data Term :: * -> * where
  Con :: a -> Term a
  Tup :: Term a -> Term b -> Term (a,b)
```

It turns out that it is surprisingly hard to make this data type an instance of almost any type class. Take the `Show` type class for instance, which represents those types which have a canonical string representation. Simplified for brevity, this is a definition of this class:

```
class Show a where
  show :: a -> String
```

It seems reasonable that `Term a` can be made an instance of `Show`, as long as `a` is an instance of `Show`.

```
instance Show a => Show (Term a) where
  show (Con x)   =
    unwords ["Con", show x]
  show (Tup x y) =
    unwords ["Tup", show x, show y]
```

However, current Haskell implementations reject this declaration. The issue resides in the second clause of the definition of `show`. In this context, the type variable `a` must stand for a tuple type, so GADT pattern matching [2] generates two fresh type variables (call them `b` and `c`, the types of `x` and `y` respectively) as well as a local type equality constraint `a ∼ (b,c)`. On the right-hand side of the equation, the *show* method is applied to $x$ and to $y$ giving rise to the wanted constraints *Show* $b$ and *Show* $c$. Here we hit a roadblock, because *Show* $(b, c)$ does not imply *Show* $b$ nor *Show* $c$ in Haskell's type system. The standard library provides a *Show* instance for pairs: **instance** $\forall a.\ \forall b.\ (\mathit{Show}\ a, \mathit{Show}\ b) \Rightarrow \mathit{Show}\ (a, b)$ **where** ..., but this is a unidirectional implication where the arrow points in the opposite direction of what we need to resolve our wanted constraints.

In this work we propose a type system extension with the aim of improving the interaction between type classes and GADTs by treating type class instance declarations as bidirectional implications.

**Background and Related Work**

As the goal of this research is to improve the interaction of type classes with other type system features, a good understanding of type classes is required. Type classes were originally introduced by Wadler and Blott [4] as a new method to support ad-hoc polymorphism.

The semantics of our solution is given by an elaboration scheme which specifies how the source language is translated into an established intermediate language, System $F_C$[3].

**Approach**

Our approach is inspired by the observation that the issue in our motivating example is caused by the unidirectionality of instance declarations. Furthermore we remark that it is easy to argue that this unidirectionality is logically unnecessary: because instance declarations are not allowed to overlap in Haskell (without extensions), instance declarations define logical bi-implications. That is, when in a given context a certain type class constraint holds, and this constraint matches the head of an instance declaration, the corresponding constraints in the instance context must also hold.

Extending Haskell with bidirectional type class instances means extending the type system and the elaboration specifications and algorithms. Extending the type system is a straightforward matter: An instance declaration of the form **instance** $\forall \overline{b}.\ \overline{Q} \Rightarrow TC\ \tau$ **where** ... gives rise to the axiom $\forall \overline{b}.\ \overline{Q} \Rightarrow TC\ \tau$ in Haskell; under our extension this declaration will also generate a list of inverse axioms $\forall \overline{b}.\ TC\ \tau \Rightarrow Q$, one for each constraint in the instance context.

Extending the elaboration specification is a somewhat more involved matter. Each type class is elaborated into a data type declaration in our target language (System $\mathrm{F}_C$), and each instance declaration is elaborated into a dictionary transformer. For example, under the dictionary passing translation scheme, the class declaration **class** $Eq\ a$ **where** $\{(==) :: a \rightarrow a \rightarrow Bool\}$ is translated into the data type declaration **data** $EqD\ a = EqD\ \{(==) :: a \rightarrow a \rightarrow Bool\}$. Instance declarations are elaborated into dictionary transformers: **instance** $Eq\ [a]$ **where** $\{(==) = \ldots\}$ becomes $eqList\ d = EqD\ \{(==) = \ldots\}$ One can view such a dictionary transformer as the "implementation" of a type class axiom: the transformer for the list instance given above can turn a dictionary for $Eq\ a$ into a dictionary for $Eq\ [a]$ for any $a$.

In the extended system, dictionary transformers must be provided for inverse instance axioms as well. That is, we must be able to construct a dictionary for $Eq\ a$ from a dictionary for $Eq\ [a]$, for example. In order to do that, we have to store in each dictionary the instance context dictionaries, but this is non-trivial because the number and type of instance context axioms varies on a per-instance-declaration basis. This makes it seemingly impossible to compose a single dictionary data type that covers all possible instance declarations (at least not without inspecting all instance declarations first).

However, we can attack this problem by using a powerful System $\mathrm{F}_C$ feature: open, non-parametric, type-level functions. In addition to the dictionary data type, our extension generates a System $\mathrm{F}_C$ *type family* for each class declaration: **type** $F_{Eq}\ a$. Furthermore, we add a *context* field to the dictionary data type, of the type $F_{Eq}\ a$. Each instance declaration gives rise to a System $\mathrm{F}_C$ axiom, which implements one of the cases of the type-level function. For our earlier list example: **axiom** $axEqList\ a : F_{Eq}\ [a] \sim a$. By type casting the element type using our new axiom, we can have our list dictionaries explicitly store the corresponding dictionary for its element type.

$$eqList\ d = EqD\ \{(==) = \ldots,\ ctx = d \rhd \mathrm{sym}\ (axEqList\ a)\}$$

Consider the following example of a function which requires bidirectional instances in order to type check:

```
f :: Eq [a] -> a -> Bool
f x = (x == x)
```

According to the elaboration specification of our extension, this function is elaborated into the following System $\mathrm{F}_C$ function:

$$f\ d\ x = (==)\ (ctx\ d\ \rhd\ (axEqList\ a))\ x\ x$$

Let us return to our motivating example, to explore how this extension addresses its issues. Recall that our wanted constraints $Show\ b$ $Show\ c$ could not be resolved from the given constraint $Show\ (b, c)$, because the $Show$ instance declaration for pairs provides only the axiom $\forall a.\ \forall b.\ (Show\ a, Show\ b) \Rightarrow Show\ (a, b)$. In our extension, this instance declaration would also provide the axioms $\forall a.\ \forall b.\ Show\ (a, b) \Rightarrow Show\ a$ and $\forall a.\ \forall b.\ Show\ (a, b) \Rightarrow Show\ b$, which suffices to resolve our wanted constraints from the given constraint.

**Results & discussion**

We have specified an extension to the Haskell programming language which lets the language treat instance declarations as logical bi-implications. More concretely, we have provided extensions to the Haskell type system and elaboration specification, as well as to type checking and elaboration algorithms. In our work, we argue that our extension

- eliminates some unpleasant surprises caused by the interaction between type classes and GADTs

- is fairly easy to implement, at least for compilers which already use System $\mathrm{F}_C$ as intermediate language (such as GHC)

- is entirely conservative: we argue that all programs written in unextended Haskell will also type check and compile under our extension with the same semantics

  A downside of our proposal is that it is incompatible with extensions which allow instance declarations to overlap. For programs with overlapping instance declarations, an implementation of our specification may produce unsound System $\mathrm{F}_C$ code.

**References**

[1] Patricia Johann and Neil Ghani. Foundations for structured programming with gadts. *SIGPLAN Not.*, 43(1):297–308, January 2008.

[2] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006.

[3] Martin Sulzmann, Manuel Chakravarty, Simon Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on types in languages design and implementation*, TLDI '07, pages 53–66. ACM, January 2007.

[4] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.