# Expelled From Class No More: Constraining Your Problem Kind

Philip K.F. Hölzenspies

University of Twente

## 1 The misbehaviour

Many Functional Programming Languages (FPLs) have some form of ad hoc polymorphism. Haskell does this through type classes. It allows for definitions and instantiations of type classes for higher-kinded types and for type classes to have multiple parameters. Since the introduction of *type families* and so-called `Constraint` kinds many more complex classes and instances have become expressible. One long-standing problem, however, has not been resolved by any of the extensions, namely the partiality of some type constructors. In this paper, we present *domain constraints* as a suggested solution for these problematic higher-kinded types and classes.

Let us first consider one of the classical instances of the problem in the Haskell library, that has been there since before the formal definition of Haskell 98. Consider the type class `Functor`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

In this class declaration, the variable in the so-called head is higher-kinded, i.e. `f :: * -> *`. This follows from its use in the function `fmap`, where it is applied to other types (`a` and `b`, which are determined by the first argument of `fmap`).

The limitation of this declaration is that `f` must be a *total* function on the domain of types, i.e. `f a` must exist for any `a`. Unfortunately, this is not always the case. To have *sets* of things, those things must be comparable for equality (otherwise there is no difference between sets and bags). In Haskell, however, the choice for the data type `Set` has been made to even require the elements to be ordered[1]. This restriction is not very visible in the definition of the type `Set`—in part because data types do not have contexts in earlier versions of Haskell and even the `DatatypeContexts` extension in GHC has been depricated because it did very little other than annotate the code for a human reader. Instead, the restriction to types of the type class `Ord` is visible in the operations on sets, e.g.

```
fromList :: Ord a => [a] -> Set a
member :: Ord a => a -> Set a -> Bool
map :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
```

[1] The `Set` type in Haskell really is a tree in order to reduce the worst-case complexity of many operations.

Since the argument to the type constructor `Set` must be a type that is an instance of type class `Ord`, we can say that the domain of the type-level function `Set` is restricted. Going back to the `Functor` type class, we point out that the type of `fmap` is unrestricted and that instances of the type class can not restrict the type of member functions. With the introduction of type families (more specifically, *associated types*) in combination with the new `Constraint` kind, this problem *seems* to have been solved. Given this alternative class declaration for `Functor`:

```
class Functor f where
  type FunctorDomConstr f a :: Constraint
  type FunctorDomConstr f a = ()
  fmap :: (FunctorDomConstr f a, FunctorDomConstr f b)
       => (a -> b) -> f a -> f b
```

we can now instantiate `Set` for this class[2]:

```
instance Functor Set where
  type FunctorDomConstr Set a = Ord a
  fmap = map
```

By giving a default implementation for the domain constraint (the empty tuple `()` denotes the empty set of constraints), instances that do not need to constrain their domain do not need to implement the associated type. This solves the problem of legacy code; anything that already was an instance of the class can remain so without needing to be rewritten or recompiled.

For this particular instance, the problem of constrained domains seems solved. Unfortunately, this solution has some critical shortcomings:

1. Type class authors must explicitly include an associated type of the kind `Constraint`. This gives rise to two sub-problems:
   (a) many authors do not do this when they have not come across the need to do it in a relevant use-case of their library;
   (b) it needlessly pollutes the namespace with names that are all specifically used for one thing;
2. It limits the validity of such class declarations to compilers that implement these extensions;
3. It does not scale well to subclasses and correctness of the propagation of these constraints in subclasses can not be verified by the compiler;

We discuss these problems in more detail in Sect. 2. The solution we propose, namely *domain constraints*, is introduced and discussed in Sect. 3. The full paper discusses implementation details.

[2] Haskell suffers the frustrating problem that types are not used to disambiguate the occurrence of different functions with the same name. Normally `map` refers to the mapping operation on lists from the standard prelude and the use of this function from `Data.Set` should be disambiguated by explicit qualification. For brevity in our code, we omit such qualifications.

## 2 The parent-teacher conference

In this section, we discuss in more detail the problems that arise with the solution that is currently possible in GHC's Haskell, using the extensions `ConstraintKinds` and `TypeFamilies`.

### 2.1 Those that play nice should not have to change

A common perspective on type classes is that they define a limited set of functionality that any instance of the class must provide. This is what the name 'ad hoc polymorphism' appeals to, i.e. the function

```
foo :: Bar a => a -> Int
```

is defined for any possible type `a`, as long as we know that at least the functionality specified by type class `Bar` is defined for it.

Consequentially, type class authors design their classes to describe a certain collection of behaviours that form a meaningful whole. Asking authors to design their classes in such a way that certain types can be more easily instantiated seems, in this perspective, to be the wrong way around. Also, what instance writers require of a class may very will vary radically between classes and instances.

### 2.2 Not all schools have deep pockets

The language extensions used are extremely powerful, but also rather complex. As discussed in Sect. 3, a much simpler language extension would suffice (and not suffer the problems discussed in this section). Although GHC is largely seen as the 'standard' Haskell compiler, it is not the only one. Other compilers implement other or fewer extensions and, at the time of writing, GHC is the only one with `ConstraintKinds` and `TypeFamilies`.

### 2.3 The most important thing is to grow

The solution as suggested above does not scale well to subclasses. As an illustration of the problem, consider the class `Applicative` as currently defined:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Any instance of `Applicative` must also be an instance of **`Functor`**. Thus, any *use* of higher-kinded type `f` must be constrained by the domain constraint from the **`Functor`** class *and* we must add the domain constraints for instances of `Applicative`. This results in a new definition of the class above, as:

```
class Functor f => Applicative f where
  type ApplicativeDomConstr f a :: Constraint
  type ApplicativeDomConstr f a = ()
  pure :: (FunctorDomConstr f a, ApplicativeDomConstr f a)
       => a -> f a
  (<*>) :: ( FunctorDomConstr f (a -> b), ApplicativeDomConstr f (a -> b)
           , FunctorDomConstr f a, ApplicativeDomConstr f a
           , FunctorDomConstr f b, ApplicativeDomConstr f b)
        => f (a -> b) -> f a -> f b
```

Of course, the default implementation of the domain constraint of this class could be that of its superclass, i.e.

```
  type ApplicativeDomConstr f a = (FunctorDomConstr f a)
```

but this only saves half of this hefty overhead and if a user does add domain constraints for this class, she must remember to explicitly include those of the superclass. There is no way to let the type-checker verify preservation of domain constraints in subclasses. Aside from the obvious undesirability of the above, the error messages become increasingly unfathomable for larger inheritance trees.

## 3 Setting boundaries

The solution we propose is a new language extension, that adds an explicit mechanism for domain constraints of higher-kinded instances of classes. We call this mechanism *domain constraints* and express it as follows. The classes remain unannotated, as is the status quo. Only at the point of instantiation do domain constraints come up. Consider the instantiation of `Set` for **`Functor`**:

```
instance Functor Set
    domconstraint (Set a) = (Ord a)
  where
    fmap = map
```

The added keyword `domconstraint` heralds a binding point, i.e. all free variables between `domconstraint` and the equals sign are bound for the right-hand side of the equals sign. The left-hand side of the domain constraint (in this case `(Set a)`) must have kind `*` and must contain precisely one type from the instance head. Said occurrence in the instance head must have kind `* -> *`. In this example there is only one type in the instance head (`Set`), but in case of multi-parameter type classes, this need not be the case. Thus, a domain constraint can only constrain the domain of one type; it does not define constrained relationships between multiple types in a multi-parameter type class.

Semantically, this adds the constraint **`Ord`** `a` for every function of the class that applies `Set` to some type `a`. These constraints are checked in the normal way. The only added requirement is that, when *domain* constraints are violated rather than the 'normal' constraints, the compiler can now generate a more informative error.

## 4 Validation

The validation of this approach by means of an implementation in GHC and the instantiation of types for a multitude of classes for which they could not previously be instantiated from hackage is work in progress.