Type Class Instances for Type-Level Lambdas in Haskell

Thijs Alkemade
Universiteit Utrecht
me@thijsalkema.de

Abstract

Haskell 2010 lacks flexibility for creating instances of type classes for type constructors with multiple type arguments. We would like to make the order of type arguments to a type constructor irrelevant to how type class instances can be specified. None of the currently available techniques in Haskell allow this satisfactorily.

To solve this, we have added the concept of type-level lambdas as anonymous type synonyms to Haskell. As higher-order unification of lambda terms in general is undecidable, we take a conservative approach to equality between type-level lambdas. We propose a number of small changes to the constraint solver that will allow type-level lambdas to be used in type class instances. We show that this satisfies our goal, while having only minor impact on existing Haskell code.

1. Introduction

1.1 Example

The first version of the unittyped package (Alkemade 2012) used a datatype similar to:

$$\textbf{data}\ \mathit{Value}\ v\ u\ d = \mathit{Value}\ v$$

A $\mathit{Value}\ v\ u\ d$ contains an object of type v and is tagged with phantom types u and d. u would represent the physical unit of the value (for example, meters, miles, seconds, etc.) and d the dimension of that unit (length, time, etc.). u and d determine what operations may be done on these values, for example, only allowing addition when the dimension of the values is the same.

After the release, a feature request asked for a *Functor* instance for *Values*. The only possible instance that the datatype would allow would give *fmap* the type:

$$\mathit{fmap} :: (a \to b) \to \mathit{Value}\ v\ u\ a \to \mathit{Value}\ v\ u\ b$$

This is not a useful instance: it can only change the dimension. Using a different dimension but the same unit breaks the invariants the library is supposed to guarantee. The desired *fmap* instance would replace the v type argument:

$$\mathit{fmap} :: (a \to b) \to \mathit{Value}\ a\ u\ d \to \mathit{Value}\ b\ u\ d$$

However, Haskell doesn't make it possible to give this instance. Eventually, all uses of the *Value* type were rewritten to use the definition:

$$\textbf{data}\ \mathit{Value}\ u\ d\ v = \mathit{Value}\ v$$

1.2 Background

A type class in Haskell is a set of polymorphic functions that can only be used on types that have instances for that class (Hall et al. 1996). This way programmers can use the same name for similar functions. This allows for more concise notation and code that uses the type class can be re-used, requiring only new instances to be written. For example, the *Eq* class makes it possible to use $\equiv$ for any type that has an instance for *Eq*, instead of requiring many different functions for checking equality.

Type classes can not only be specified for normal types of kind $*$, but also for types of other kinds, in particular those of an arrow kind like $* \to *$. Every type class requires its instances to have a specific kind, determined by how many arguments the type variable receives in the function signatures of the type class (Peterson and Jones 1993).

For example, *Eq* has a type variable of kind $*$, as the type variable a in the class head does not receive any arguments:

$$\begin{array}{l}\textbf{class}\ \mathit{Eq}\ a\ \textbf{where}\\ \quad (\equiv) :: a \to a \to \mathit{Bool}\end{array}$$

The *Functor* class has a type variable of kind $* \to *$, as f is used with one argument (as a and b are of kind $*$):

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}\\ \quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b\end{array}$$

Now consider a type class definition using a type of kind $* \to * \to *$, such as *Category* (Figure 3) or *Arrow* (Figure 4). Suppose we want an instance of the class *Functor* on the first type argument of the *Arrow*. Such an instance is tedious to obtain using the current features of Haskell. In this paper we show how to add type-level lambdas to instance heads in Haskell, making it possible to write, for instance:

$$\begin{array}{l}\textbf{data}\ T\ x\ y\ z = T\ y\\ \textbf{instance}\ \mathit{Functor}\ (\Lambda y\ .\ T\ x\ y\ z)\ \textbf{where}\\ \quad \mathit{fmap}\ f\ (T\ y) = T\ (f\ y)\end{array}$$

```
*Main> fmap (+1) (T 42)
T 43
```

1.3 Contents

In Section 2 we will establish some preliminaries about applied type variables. Section 3 will describe the problem in detail. Section 4 explains the current potential solutions to this problem and their downsides. Section 5 will formalize the notion of a type-level lambda. Section 6 gives a general background about type checking and constraint solving in Haskell and Section 7 will explain the changes to support type-level lambdas. In Section 8 we will show what is possible with these changes and in Section 9 we describe some potential problems with other GHC extensions.

2. Preliminaries

First, we introduce some preliminaries. In Section 2.1 we cover how higher-kinded types can be constructed and in Secion 2.2 where applied type variables are used in Haskell.

2.1 Higher-kinded types

Haskell has two flavors of types: variable and constants. Type variables always start with lower-case letters. Type constants are type constructors, which are the types introduced by **data** and **newtype** definitions and always start with an upper-case letter.

Type constructors can take type arguments which make them into type-level functions. By not providing all type arguments, type constructors can be made into type constants of a higher kind.

Constant types with different orderings of type arguments are isomorphic. We can show this by applying a type-level curry, and using the isomorphism between (a, b) and (b, a):

$$
\begin{aligned}
T\ x\ y &\text{ is isomorphic to } T\ (x, y)\\
T\ (x, y) &\text{ is isomorphic to } T\ (y, x)\\
T\ x\ y &\text{ is isomorphic to } T\ y\ x
\end{aligned}
$$

Thus, the order of the type arguments of a type constructor may appear irrelevant. However, the exact order is important when considering which higher kinded types can be formed. For example, a type constructor with three arguments (**data** $T\ x\ y\ z$), allows only the following types of higher kinds to be formed:

$$
\begin{array}{ll}
T & :: * \to * \to * \to *\\
T\ x & :: * \to * \to *\\
T\ x\ y & :: * \to *\\
T\ x\ y\ z & :: *
\end{array}
$$

This misses, for example, $T\ \cdot\ y\ z$ of kind $* \to *$, where a type argument is used as the x.

2.2 Redundant applied type variables

The *Functor* class uses types consisting of a type variable applied to another type variable: $f\ a$ and $f\ b$. This means that f can be substituted by any type of kind $* \to *$ and a and b by any type of kind $*$. Lets look at the situations where a type like $f\ a$ is useful.

Haskell programmers often try to find the most general type for a function, because using the most general type means the function can be used in more situations. This means that when a type variable applied to another type variable is used (like $f\ a$), it is very likely at least one of the following holds: Either f occurs somewhere else in the type, in a class constraint on f or f applied to a different type or type variable, or a occurs somewhere else in the type.

If neither condition holds (which means that only $f\ a$ is used, no separate f or a), then the function could be given a more general type by substituting $f\ a$ by a single, new type variable. For example, consider the type signature:

$$fun :: f\ a \to b \to f\ a$$

There are only two possible implementations of a function with this type:

$$
\begin{aligned}
&fun\ x\ y = x\\
&fun\ x\ y = \bot
\end{aligned}
$$

For both options $c \to b \to c$ would be a valid type too, and this is in fact the most general type for *fun*. It does mean the function can be called on more types: the original type signature would not allow *fun* `'a'`, as `'a'` :: *Char* and $Char \not\sim f\ a$. However, it is very unlikely a programmer would want to write a function that can only called on values with an applied type, without actually making use of the type application. For example, without any class constraint on f it would be impossible to write a function of type $f\ a \to f\ b$ or $a \to f\ a$.

```
class Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
```

Figure 1. The *Monad* class.

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

Figure 2. The *Applicative* class.

```
class Category cat where
  id :: cat a a
  (∘) :: cat b c → cat a b → cat a c
```

Figure 3. The definition of *Category* from `Control.Category`.

```
class Category a ⇒ Arrow a where
  arr    :: (b → c) → a b      c
  first  :: a b c   → a (b, d) (c, d)
  second :: a b c   → a (d, b) (d, c)
  (***)  :: a b c   → a b′     c′
                    → a (b, b′) (c, c′)
  (&&&)  :: a b c   → a b      c′
                    → a b      (c, c′)
```

Figure 4. The *Arrow* class.

3. Problem description

Although the order of the arguments of a type constructor may appear irrelevant for starting Haskell programmers, they influence the instances of higher-order type classes that can be specified. Some examples:

- An example datatype $T\ x\ y\ z$ can only be a *Functor* that modifies z, or an *Applicative* that applies a function to z, or a *Monad* that can lift a value given a value of type z into the monad, etc.

  So for any type with at least one type variable, the last type variable is the variable that *Functor*, *Monad* (Figure 1) and *Applicative* (Figure 2) act on.

- Instances for *Category* (Figure 3) and *Arrow* (Figure 4) use the two last variables. In particular, this means that in any instance of *Arrow* that is also an instance of *Functor*, *fmap* will always modify the second variable of the *Arrow* (which would be the result type, in the usual *Arrow* $(\to)$).

  So $T\ x\ y\ z$ can only be made an *Arrow* where *arr* has type $(y \to z) \to T\ x\ y\ z$. We can not make T an arrow where x and y are the source and destination of the arrow, but where z is the variable that is modified by *fmap*.

- The *MonadTrans* class from the mtl package (Gill 2006-2012) and the *MFunctor* class from the mmorph package (Gonzalez

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a
class MFunctor t where
  hoist :: (Monad m) ⇒ (∀ a . m a → n a)
                    → t m b → t n b
```

Figure 5.

2013-2014) also use the last two variables, see Figure 5. However, they both expect the first variable to have kind $* \rightarrow *$ and the second one to have kind $*$.

But *Category* and *Arrow* require the two type variables to be of equal kind. So it's never possible to give a type both an instance for *MonadTrans* or *MFunctor* and an instance for *Category* or *Arrow*.

So defining new higher-order type classes has the following problem: the order of the arguments of the type class variable determines, for every instance of this new class, how it can be instantiated as a *Functor*, *Arrow*, etc.

Suppose f is expected to have two type arguments, x and y, in the following new class:

```
class C f where
  foo :: f ? ?
```

Then there are two options: using $f\ x\ y$ and using $f\ y\ x$. Which of these two is used determines for all instances of C how they can be made a *Functor*, *Arrow*, etc. There might be a logical choice (when the class has some relationship with *Arrow* or *Functor*), but there's no way the class can leave it up to the programmer to make a choice per type. These restrictions make it difficult to introduce new higher-order type classes and may lead to confusing ordering of type arguments for complicated type constructors.

The goal of this paper is to make the order of the arguments of a type constructor irrelevant. In particular, they should have no effect on how instances of higher-order classes can be defined. We would like to do this under the following conditions:

1. The type checker requires no user-added type signatures where they are currently not required.
2. It should not be required to duplicate functions or type classes or to make large changes to existing functions or type classes.

For example, we would like to specify instances like in Section 1. There, the type constructor has three arguments and we would like to be able to specify a *Functor* instance where *fmap* affects the first one, i.e., has the type $(a \rightarrow b) \rightarrow T\ a\ y\ z \rightarrow T\ b\ y\ z$.

4. Existing solutions

Currently, the simplest solution to obtain the correct instances for a type would be to first consider the type classes that a type should have instances for, and then order the type variables accordingly. If the type variables of existing code do not match the order required for the desired class instances, they can be reordered to match by rewriting the type everywhere it is used.

For example, if $T\ x\ y\ z$ later needs to have *Functor* instance that works on x, that would mean replacing every $T\ x\ y\ z$ with $T\ y\ z\ x$. In a large codebase, this could be a significant amount of work.

4.1 Type Synonym Instances

The GHC extension `TypeSynonymInstances` (The GHC Team 2013) may appear as a good solution. This extension allows type synonyms in the head of an instance declaration. Without this extension, only newtypes and datatypes can be used in the class instance heads.

```
data T x y z = T x
type TFunctor y z x = T x y z
instance Functor (TFunctor y z) where
  fmap :: (a → b) → T a y z → T b y z
```

Figure 6. Using `TypeSynonymInstances`, this is how one could try specifying an instance for *Functor* using a type synonym for T, but GHC will reject this.

```
class Functor2 f where
  fmap2 :: (a → b) → f a x → f b x
class Functor3 f where
  fmap3 :: (a → b) → f a x y → f b x y
```

Figure 7. Extra *Functor* instances for different variables.

By creating a type synonym that orders the type variables in the way they should be used by that class, the desired type for the instance could be specified. See for example Figure 6. Here a new type synonym is created for T which specifies the order of the type variables for its *Functor* instance.

As nice as this may seem, it will not be accepted by GHC: `TypeSynonymInstances` only allows fully applied type synonyms in the instance head. A type synonym can't be partially applied to form a type of kind $* \rightarrow *$ and supplied to *Functor*.

The only way a type synonym can be used to represent a type of kind $* \rightarrow *$ (or higher), is when the rhs of the type synonym already has kind $* \rightarrow *$:

```
data T x y z = T x
type TFunctor y x = T x y
```

But this means we are back to supporting only the limited set of higher-kinded types seen in Section 2.1. Therefore this solution does not meet our main goal.

4.2 More type classes

A simple solution would be to add more type classes. For every ordering of type variables a programmer might want to use for a class, a new copy of the class can be added. See for example Figure 7, where *Functor2* and *Functor3* are defined which act on the second and third to last type variable, respectively. The `Bifunctors` package (Kmett 2011-2013) uses this approach: it allows types to be specified as functors on both the last and the second to last variable at the same time.

The advantage of this solution is that instances for *Functor2* and *Functor3* do not overlap. For every variable of a type constructor it would therefore be possible to indicate whether it allows an *fmap*$[n]$.

There is however a serious disadvantage to this solution: every function with a *Functor* constraint needs to be copied for *Functor2*, *Functor3*, etc. The implementation will be the same, except for the use of *fmap2*, *fmap3*, etc. instead of *fmap*. See Figure 8. The main goal of type classes is to avoid code duplication, but this solution adds code duplication.

Another disadvantage to this solution is that the number of extra type classes increases fast, especially for even higher-order type classes. For example, the *Category* class requires type constructors of kind $* \rightarrow * \rightarrow *$ (Figure 3). Every possible pair would require

```
increase :: (Functor f) ⇒ f Int → f Int
increase = fmap (+1)
increase2 :: (Functor2 f) ⇒ f Int x → f Int x
increase2 = fmap2 (+1)
increase3 :: (Functor3 f) ⇒ f Int x y → f Int x y
increase3 = fmap3 (+1)
```

Figure 8. Every function using a *Functor* constraint needs to be copied for *Functor2*, *Functor3*, etc.

```
newtype Flip t b a = Flip {unFlip :: t a b}
```

Figure 9. *Flip* from `TypeCompose`.

```
data T x y z = T y
instance Functor (Flip (T x) z) where
  fmap :: (a → b)
     →  Flip (T x) z a
     →  Flip (T x) z b
  fmap f (Flip {unFlip = T y})
          = Flip {unFlip = T (f y)}
```

Figure 10. Using *Flip* to specify flipped *Functor* instances.

```
newtype Flip2 t c b a   = Flip2 {unFlip2 :: t a b c}
newtype Flip3 t d b c a = Flip3 {unFlip3 :: t a b c d}
```

Figure 11. Generalizations of *Flip*.

a separate class, *Category_2_3*, *Category_1_3*, *Category_2_1*, etc.

While this solution meets our main goal, it does not satisfy condition 2: existing functions using type classes can not be reused for the newly introduced type classes.

4.3 Newtype wrappers

The `TypeCompose` package (Elliott 2007-2013) contains the newtype definition in Figure 9. It can be viewed as a type-level variant of *flip*: the last two type arguments of the *Flip* type are the last two type arguments of the wrapped type, but reversed. This also makes it possible to write instances where the last two variables are reversed, as can be seen in Figure 10. It's not only possible to flip the last two arguments, but *Flip* can be generalized to every reordering of type variables, see Figure 11.

The difference with `TypeSynonymInstances` is that *Flip*s are newtypes, not type synonyms. Therefore the type on the instance head is different. This also means that instances for different variants of *Flip* will not overlap. So also here it is possible to specify, for every type variable, whether the type has a *Functor* over that variable or not.

One disadvantage of this solution is that every argument of the relevant type needs to be wrapped in with *Flip*, and unwrapped with an *unFlip* call. See Figure 12 for how *fmap* would need to be called for a *Flip*ped type.

This solution meets our main goal, however, the extra boilerplate code to wrap and unwrap datatypes before and after applying type class functions does not satisfy condition 2.

4.4 Associated type families

The reason the *Functor* class needs instances with a variable $f :: * \to *$ is to make it possible to construct $f\ a$ and $f\ b$.

```
fmap (+1) (T 42)
⇓
unFlip (fmap (+1) (Flip (T 42)))
```

Figure 12. Wrapping and unwrapping *Flip*ped *Functor* instances.

```
class Functor f where
  type FunctorApp f c :: *
  fmap :: (a → b) → FunctorApp f a
                  → FunctorApp f b
instance Functor (Maybe x) where
  type FunctorApp (Maybe x) y = Maybe y
  fmap :: (a → b) → FunctorApp (Maybe x) a
                  → FunctorApp (Maybe x) b
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
data T x y z = T x
instance Functor (T x y z) where
  type FunctorApp (T x y z) v = T v y z
  fmap :: (a → b) → FunctorApp (T x y z) a
                  → FunctorApp (T x y z) b
  fmap f (T x) = T (f x)
```

Figure 13. *Functor* with a associated type family *FunctorApp*.

It is not vital for *Functor* that f is a type constructor and a the last argument, the only part that matters is that $f\ a$ is a type that contains a somewhere, and $f\ b$ the same type but with a replaced by b. However, it is currently not possible to express that this a is anywhere, not just on the last position.

With the `TypeFamilies` language extension of GHC (The GHC Team 2013) it is possible to define type families within type classes. We can use such an associated type family instead of using $f\ a$ and $f\ b$ directly. This way, the type family indicates how the types are changed within the class functions, see for example Figure 13 of how this can be used for *Functor*. Note that *Functor* no longer receives a type of kind $* \to *$ but of kind $*$, as it doesn't need to be apply it: it uses the *FunctorApp* definition for that instead.

The type family would have one argument for the type of the instance and one argument for every variable the type class uses. The rhs of the type family should be the first type, with the arguments substituted at the right positions.

While this may seem like a lot of extra code for all instances and classes, it would be possible to translate definitions written with the current syntax to this format automatically. Only for instances where the extra expressiveness is needed the type families would need to be added by hand.

However, this solution has a problem with the way type families work currently: to call *fmap* $(+1)$ on *Just* 1, the type family equality constraint $\mathit{FunctorApp}\ f\ a \sim \mathit{Maybe}\ \mathit{Int}$ would need to be solved (a substitution for f needs to be found). However, type families in GHC are not injective. There could be other types T which defined $\mathit{FunctorApp}\ T\ x = \mathit{Maybe}\ x$, although this is completely bogus. There is no way to prohibit this, so the equality constraint can not be solved. This makes type families as they currently work unusable for this goal.

This solution does not satisfy our main goal: the non-injectivity of the associated type families makes it impossible to use the type class functions.

```
data T x y z = T x y
instance Functor (Λx . T x y z) where
  fmap f (T x y) = T (f x) y
instance Functor (Λy . T x y z) where
  fmap f (T x y) = T x (f y)
```

```
*Main> fmap (+1) (T 2 3)
```

Figure 14. Resolving which instance to use would be impossible without a type signature or extra code.

4.5 Conclusion

None of the mentioned solutions satisfies the main goal and all conditions in Section 3.

A number of the possibilities allow multiple instances per type constructor for a given type class. For example, some allow defining *Functor* instances (or a new instance meant to look like *Functor*, like *Functor2*) for both the last and the second to last type variable. Although this can be useful, it can not satisfy conditions 1 and 2 at the same time: without either type annotations or boilerplate code, the compiler would be unable to determine which instance to use for an *fmap* call. See for example Figure 14. We shall therefore consider multiple instances for the same type overlapping, even when they use a different ordering of type variables.

5. Type-level lambdas

In the previous sections we have informally used Λ to denote a type-level lambda function. In this section we will give a precise definition and we will investigate the issues it can create with type checking.

5.1 Undecidability

Lets examine more closely why the solution in Section 4.1 is rejected by the compiler. Type synonyms can have zero or more arguments. However, contrary to type families they can not do any case distinction on those variables. We can consider type synonyms polymorphic "functions" on types. This means that determining whether two partially applied type synonyms are equivalent comes down to determining whether two lambda expressions are equivalent. This problem is known as unification: we need to find a substitution of the free variables in the two expressions such that they become equivalent. Specifically, it is higher-order unification: free variables may be substituted by new lambda abstractions. Higher-order unification is undecidable in general, as was shown by (Huet 1973).

Haskell therefore uses the rule that type synonyms must be fully applied before they may be used as a type. That's why the example in Section 4.1 is forbidden: it is using a partially applied type synonym on the place of a type.

5.2 Adding Type-level Lambdas

To write an instance of *Functor* where *fmap* has type $(a \to b) \to T\ a\ y\ z \to T\ b\ y\ z$, the instance head would need to have a type τ such that $\tau\ a$ is equal to $T\ a\ y\ z$ and $\tau\ b$ is equal to $T\ b\ y\ z$. Type synonyms can not be used here currently, but even if they could, it would be more convenient to have notation that does not require defining new type synonyms for every type class and type constructor. Therefore we introduce as new notation the "type-level lambda": $\Lambda x\ .\ T\ x\ y\ z$ is a type where x is bound by the lambda, and y and z are free.

Evaluation is, just like value-level λ-functions, a β-reduction step where the argument is substituted for a variable:

$$(\Lambda x\ .\ M)\ y \to_\beta M[x := y]$$

$$\frac{\mathcal{Q}; Q; \Gamma, x : \kappa \vdash T : \kappa'}{\mathcal{Q}; Q; \Gamma \vdash \Lambda x.T : \kappa \to \kappa'}$$

Figure 15. Type-level lambda kind-checking rule. $\mathcal{Q}$ is a top-level environment and Q is a set of constraints. Γ is a kinding environment, T is a type and κ and κ' are kinds.

```
newtype (Monad m) ⇒ MaybeT m a
    = MaybeT {runMaybeT :: m (Maybe a)}
```

Figure 16. The $MaybeT$ monad transformer.

The kind checking rule that applies to type-level lambdas can be found in Figure 15.

In dependently typed languages type-level lambdas and value level lambdas are the same concept, but also in other, non-dependently typed functional languages the concept exists, for example in Scala, see 10.2. Although type-level lambdas do not exist in Haskell itself, they do occur in the typed internal representation of GHC called "Core".

Note that $\forall\ x\ .\ T\ x\ y\ z$ and $\Lambda x\ .\ T\ x\ y\ z$ do not mean the same thing. $\forall\ x\ .\ T\ x\ y\ z$ has the same kind as $T\ x\ y\ z$, but $\Lambda x\ .\ T\ x\ y\ z$ has kind $l \to k$, with $x :: l$ and $T\ x\ y\ z :: k$.

5.3 Decidable unification of type-level lambda terms

We can see type-level lambdas as anonymous type synonyms, just like value level lambda functions as anonymous functions. Unification of type-level lambdas will therefore be undecidable in general. We can use multiple solutions for this:

1. Use the same limitation as for type synonyms: type-level lambdas need to be fully applied before they may be used as a type, with an exception for instance heads.

   A disadvantage to this solution is that, for example monad transformers (see Figure 16), which take a type variable with a *Monad* constraint, could not be specified for monads which use type-level lambdas. For example, $MaybeT\ (\Lambda x\ .\ T\ x\ y\ z)\ a$ would be forbidden.

2. We can try to limit the unification problems to a subset that is decidable.

   Lower-order unification is decidable, however, not sufficient for our goal: it would be unable to ever unify $f\ a$ with anything.

   Higher-order matching is also decidable. Matching is unification where one of the two arguments is closed. However, this is also not useful: type class instances do not need to be closed, they may contain free type variables.

3. Use Guided Higher-Order Unification (Λ_{GHOU}) as proposed by (Neubauer and Thiemann 2002). The restrictions applied to Λ_{GHOU}, compared to general higher-order unification are:

   - Projections are ruled out. These are lambda abstractions where the body of the type starts with one of the abstracted variables. This forbids lambda types of the form:

     $\Lambda x\ .\ x\ ...$

   - The types are restricted to a subset of the lambda calculus called λI: this only allows abstracted variables that occur free in the body of the type. This forbids, for example:

$\Lambda x\ .\ Int$

4. We consider these solutions either too restrictive or too complicated. Instead, we chose to do the following:
   - Type-level lambda functions may be used unapplied. However, they are not unified: they must be α-equivalent or type checking will fail. This means they may be used in monad transformers, like in case 2, but they must be used consistently.
   - We make an exception for unifying with instance heads. This will be covered in Section 7.2.

6. Type checking Haskell

In the this section, we will give a short description of the type checking and constraint solving for Haskell that is documented in (Vytiniotis et al. 2011).

Type checking is divided into two phases: first type inference, where constraints are generated, then these constraints are solved. The solution of the solving of these constraints is a set of substitutions and (possibly empty) set of unsolved constraints.

6.1 Example

Consider the following source:

$f\ x = x + 5$

The type inferencer will start out by giving f a function type, as it is specified with one argument. x is its argument, so it gets assigned the argument's type.

$f :: \alpha \to \beta$
$x :: \alpha$

By looking up the Num class, it will determine that $(+) :: (Num\ \ a) \Rightarrow a \to a \to a$ and $5 :: (Num\ \ b) \Rightarrow b$. By looking at the body of f and how it uses $(+)$, the type inferencer will introduce the constraints $\alpha \sim a$, $b \sim a$ and $\beta \sim a$. So the type inferencer will end with the set of constraints $(\alpha \sim a, b \sim a, \beta \sim a, Num\ a, Num\ b)$.

The constraint solver can turn all these equality constraints into substitutions, as they are simple. The result is the substitution $[\alpha \mapsto a, \beta \mapsto a, b \mapsto a]$. Due to the substitution, the constraint $Num\ \ b$ has become unnecessary, as it is equal to $Num\ \ a$, so only one constraint is left, $Num\ \ a$. This constraint is left over, which means it gets added to f's type signature, making the final type signature $(Num\ \ a) \Rightarrow a \to a$.

6.2 Generating constraints

The important constraints to consider are:

1. **Class constraints:** a class followed by zero or more types:

   $D\ x$

   In this paper we will only look at classes using exactly one variable. See also Section 9.1.
2. **Equality constraints:** constraints demanding two types are equal:

   $a \sim b$

   We can consider different types of equality constraints:

   (a) **Impossible:** Equality constraints which contain different concrete types on both sides can not be solved:

   $Char \sim Int$

   This also includes equality constraints where an applied type is matched with an non-decomposable type:

   $f\ a \sim Int$

   (b) **Simple:** Equality constraints which contain a single type variable on one side:

   $a \sim T$

   (c) **Type family:** Type families are the only equality constraints that might need to be specified by the programmer manually:

   $F\ a \sim T$

   (d) **Applied:** Equality constraints which have an applied type variable on one side, and a different applied type variable or a concrete type on the other side:

   $f\ a \sim g\ b$
   $f\ a \sim [Int]$

6.3 Solving constraints

To solve the generated constraints, a number of different solvers are applied one after another in a loop. If during one iteration of the loop no changes are made to the set of remaining constraints, the loop terminates and the set of constraints that are left is returned. If this set is non-empty, then these are usually turned into errors.

The different solvers include:

1. **Canonicalization:** Before constraints are passed to other solvers, they are canonicalized. This makes the constraints simpler and ensures that constraints are always following certain rules. For example, constraints that fail the occurs check are rejected, nested type families or type families in class constraints are turned into flat type families (by introducing fresh type variables) and applied equality constraints are split into simple equality constraints.

   We will use `[W]` to denote wanted constraints and `[G]` to denote given constraints. Some examples of the canonicalization step:

   $\{\texttt{[W]}\ f\ a \sim g\ b\} \to \{\texttt{[W]}\ f \sim g, \texttt{[W]}\ a \sim b\}$
   $\{\texttt{[W]}\ f\ a \sim [Int]\} \to \{\texttt{[W]}\ f \sim [\,], \texttt{[W]}\ a \sim Int\}$

   This is allowed because f and g have to be (partially applied) type constructors: they can't be a type synonym or type family. As seen in Section 2.1, every type constructor has at most one partially applied type for a given kind so it can be unambiguously resolved.
2. **Binary interaction:** Another solver looks at 2 canonical constraints together. For example, a simple equality constraint and another constraint will apply the equality constraint as a substitution on the other constraint. Having two identical constraints means one of them can be deleted. Type family constraints with identical lhs, but different rhs generate an equality constraint between the rhs and allow one of the two type family constraints to be deleted. The binary interaction rules only look at two constraints that are either both given, or both wanted.

   Here are some examples of the binary interaction step:

   $\{\texttt{[W]}\ Num\ a, \texttt{[W]}\ Num\ a\} \to \{\texttt{[W]}\ Num\ a\}$
   $\{\texttt{[W]}\ a \sim T, \texttt{[W]}\ Num\ a\}$
   $\quad \to \{\texttt{[W]}\ a \sim T, \texttt{[W]}\ Num\ T\}$
   $\{\texttt{[W]}\ F\ Int \sim [a], \texttt{[W]}\ F\ Int \sim [Int]\}$
   $\quad \to \{\texttt{[W]}\ [a] \sim [Int]\}$
3. **Simplification:** The simplifier also looks at 2 canonical constraints, but specifically pairs of constraints where one of them

is given (i.e., given by the user by supplying a type signature) and the other is wanted (i.e., generated during type checking).

Obviously, a given constraint and a wanted constraint that are identical means the wanted constraint can be deleted. Given simple equality constraints can also be used as substitutions on wanted constraints.

Some examples:

$\{\, \texttt{[W]}\ \mathit{Functor}\ f, \texttt{[G]}\ \mathit{Functor}\ f \,\} \rightarrow \{\, \texttt{[G]}\ \mathit{Functor}\ f \,\}$
$\{\, \texttt{[W]}\ \mathit{Functor}\ f, \texttt{[G]}\ f\ \sim\ g \,\}$
$\quad \rightarrow \{\, \texttt{[W]}\ \mathit{Functor}\ g, \texttt{[G]}\ f\ \sim\ g \,\}$

4. **Top-level interaction:** During top-level interaction, the instances of type classes and type families are used to solve wanted type class and type family constraints, respectively. This may introduce new constraints, for example for superclasses of instances.

   For example, suppose the usual $\mathit{Functor}\ [\,]$ instance is in scope:

   $\{\, \texttt{[W]}\ \mathit{Functor}\ [a] \,\} \rightarrow \{\,\}$

   Suppose we have a type family:

   $\textbf{type family}\quad F\ x\quad ::\ *$
   $\textbf{type instance}\ F\ \mathit{Int} = [\mathit{Int}]$

   Then the top-level interaction step will do the following:

   $\{\, \texttt{[W]}\ a\ \sim\ F\ \mathit{Int} \,\} \rightarrow \{\, \texttt{[W]}\ a\ \sim\ [\mathit{Int}] \,\}$

7. Adding Type-Level Lambdas to GHC

We have set out to include our proposed changes in GHC. Our development started off with the 7.7 version of GHC, which is the development branch that was later released as GHC 7.8.

These changes consist of 3 parts: the parser is modified to allow notation with /\ for type-level lambdas, the internal representation of types is modified to support Λ and the constraint solving is adapted to take into account the possibility of type-level lambdas in class instance heads.

/\ is currently valid syntax for a term-level operator. However, because this is type-level syntax it will not cause problems. With the GHC extension `TypeOperators` (The GHC Team 2013), /\ can be used as a valid type operator. We assume that because using it requires an extension, there will not be many problems with existing code already using this syntax.

7.1 Parser

The changes to the parser are simple: /\ follows the same rules as `forall`: /\ must be followed by one or more types, which can optionally have a kind signature when `KindSignatures` is set.

For example:

```
/\ a . [a]
/\ x . ()
/\ (f :: * -> *) . f Int
```

7.2 Evaluation of type-level lambdas

We evaluate type-level lambdas greedily: when a type-level lambda is encountered which is applied to an argument, the substitution is carried out immediately. We shall show that this can not introduce non-termination in the type checker.

The type-level language of Haskell can be considered a "typed" lambda calculus, where Haskell's kinds form the types. The kinds form a simply-typed lambda calculus, thus the types are strongly normalizing. This means that we can not write non-terminating combinators from the untyped lambda calculus, such as Ω:

$$\omega = (\lambda x.x\ x)$$
$$\Omega = \omega\ \omega$$

The ω combinator can not be type checked as its type fails the occurs check: suppose $\omega\ ::\ \tau \rightarrow \sigma$, then $\sigma = \tau\tau$, which means $\tau = \tau \rightarrow \sigma$. This equation can not hold for types. Thus Ω also fails to type check. In GHC, trying to define ω would cause a "Kind occurs check" error.

Another way we can achieve non-termination is through recursion. Haskell **let**-bindings can refer to themselves, which can lead to infinite recursion.

It is impossible to create a recursive definition from lambda functions alone: they are anonymous, so can not refer to themselves. It would be possible with the Y combinator, but a type-level Y combinator is impossible for the same reason as Ω: it fails the kind occurs check, in the same way that the Y combinator, without using newtypes, fails the type occurs check in Haskell.

But, just like how **let**-bindings in Haskell allow terms to be named, we can give names to type-level lambdas by defining a type-level function for it. Haskell currently knows two different forms of type-level functions:

- Type synonyms
- Type families

Type synonyms can't be recursive: trying to do recursion or mutual recursion in type synonyms will give an error "Cycle in type synonym declarations".

Type families can be (mutually) recursive, but only when the `UndecidableInstances` (The GHC Team 2013) extension is enabled. When this extension is not enabled, recursion will be forbidden because it will mean the equation has a rhs that does not follow the rules which require the rhs to be "smaller" than the type family arguments.

Turning on `UndecidableInstances` will cause GHC to lift many of its restrictions that should guarantee termination. With this flag on, it is already possible to cause infinite loops in the type checker, so the addition of type-level lambdas does not change that.

7.2.1 Type-level lambdas in instances

The main goal of our work is to allow type-level lambdas in the heads of type class instance declarations. To avoid problems with undecidability here, we will allow only well-formed type-level lambdas as instance heads.

DEFINITION 1. *A* well-formed *lambda function is a lambda function that can can be constructed using the following grammar:*

```
L := /\ x . L
   | T
```

where `T` *are the type constructors possibly applied to some types, under the extra condition: every variable* `x` *that is bound by a* /\ *must occur exactly once as an argument to the inner type constructor.*

In other words, a type-level lambda function is well-formed if every lambda bound type variable is used exactly once, and the body of the lambda is either again a type-level lambda, or starts with a type constructor.

See Figures 17 and 18 for examples of well-formed and non well-formed types.

The advantage of only allowing well-formed type-level lambda functions is that their unification is simple, which avoids the undecidability involved with higher-order unification.

$\Lambda x \, . \; T \; x \; y \; z$
$\Lambda x \, . \; [x]$
$\Lambda x \, . \Lambda y \, . \Lambda z \, . \; T \; z \; x \; y$

Figure 17. Well-formed types.

$\Lambda x \, . \; T \; [x] \; y \; z$
$\Lambda x \, . \; T \; x \; x \; z$
$\Lambda x \, . \; [\mathit{Int}]$

Figure 18. Not well-formed types.

For most instances given by programmers the well-formedness restriction should not cause problems: it allows reordering of type variables (as we intended to do), but no more complicated type-level functions. In this sense reordered instances are just as powerful as instances which can be written without this extension.

Some examples of instances that can not be written as they use non well-formed type-level lambdas:

- Definition 1 states that the body of a type-level lambda must start with either another type-level lambda, or a type constructor. Notably, it may not start with a type variable, in particular the lambda-bound type variables.

  This means for example that an instance using $\Lambda x \; . \; x \; \mathit{Int}$ can not be specified. However, this type would have kind $(* \to *) \to *$ (note that this is not the same as $* \to * \to *$). Classes using type variables of this kind, or even higher kinds, are quite rare, at least for now.

- A type-level lambda bound type variable must occur as a direct argument to the inner type constructor. It may not be inside another type constructor in the argument.

  This means that, for example, $\Lambda x \; . \; \mathit{Maybe} \; [x]$ is not well-formed. We believe this will not be a problem for Haskell programmers, as type class arguments are currently always interpreted to refer to one of the direct arguments of a type constructor.

- A type-level lambda bound variable must occur exactly once as an argument to the inner type constructor.

  For example, $\mathit{Functor} \; (\Lambda x \; . \; (x, x))$ is not well-formed. Just like the previous case, we believe this will not be a problem for Haskell programmers, as type class arguments are currently always interpreted to refer to exactly one of the direct arguments of a type constructor, never multiple at the same time.

- From a category theoretic point of view it might be interesting to define the identity functor and functor composition. We can express these with type-level lambdas as can be seen in Figure 19. However, this will not work, as both instances' heads are non well-formed.

  Although they are interesting, they are not very useful in Haskell. First of all, with the way instances are resolved in Haskell the identity functor would overlap with every other possible *Functor* instance. When the user would call *fmap f* with the intention to use the identity functor (i.e., on a type with no other functor instance), $f \; x$ can do the same. Secondly, when using *fmap* on composed functors, GHC could not resolve whether the call to *fmap f* is meant to apply on the first functor alone, or on the composition. $\mathit{fmap} \; (\mathit{fmap} \; f)$ can be used instead to apply to the composition.

  Alternatively, it is possible to use the wrappers *Identity* and *Compose* from the `transformers` package (Gill and Paterson 2009-2012) to obtain identity functors and functor composition.

instance $\mathit{Functor} \; (\Lambda x \; . \; x)$ **where**
  $\mathit{fmap} \; f \; x = f \; x$
instance $(\mathit{Functor} \; f, \mathit{Functor} \; g)$
  $\Rightarrow \mathit{Functor} \; (\Lambda x \; . \; f \; (g \; x))$ **where**
  $\mathit{fmap} \; f \; x = \mathit{fmap} \; (\mathit{fmap} \; f) \; x$

Figure 19. These instances contain non well-formed types.

7.3 Constraint solving

We can now express type class instances with type-level lambda instances, but to be able to use them the constraint solver needs to be able to find and use those instances. This does not, however, require many changes to the class constraint solver. The changes are mostly in the solving of equality constraints, specifically applied equality constraints.

Firstly, the splitting of applied equality constraints as happens during canonicalization (Section 6.3) is no longer allowed. Suppose the following data type exists, and in the code *fmap* is applied to it:

data $T \; x \; y \; z = T \; x \; y \; z$
$\mathit{foo} = ...\mathit{fmap} \; g \; (T \; 1 \; () \; \texttt{'a'}) \; ...$

During type inference, the constraint $f \; a \; \sim \; T \; \mathit{Int} \; () \; \mathit{Char}$ will be created. However, we do not yet know how which *Functor* instance for T exists. The possible decompositions are therefore:

- $f \; \sim \Lambda b \; . \; T \; b \; y \; z$ and $a \; \sim \; \mathit{Int}$
- $f \; \sim \Lambda b \; . \; T \; x \; b \; z$ and $a \; \sim \; ()$
- $f \; \sim \Lambda b \; . \; T \; x \; y \; b$ and $a \; \sim \; \mathit{Char}$

The constraint can not be split, but it has to be solved as a whole. In Section 7.3.1 we will set the requirements for how these constraints can be split (an applied type variable on one side, a concrete type on the other) and in Section 7.3.2 for the case where both sides consist of an applied type variable.

7.3.1 Applied-Concrete

Suppose a constraint $f \; a \; \sim \; T \; x \; y \; z$ is encountered: an applied type variable on one side, with a concrete type on the other side.

As mentioned in Section 4.5, satisfying all the conditions from Section 3 at once can only be done when every type still allows only one instance per type class. We shall therefore keep this restriction and use it to solve these constraints. When an applied equality constraint is encountered with a concrete type on the rhs, and a type class is known that applies to that concrete type, then we use the type-level lambda used in the class instance to pick the correct decomposition of the equality constraint.

For example, suppose the following are given:

- The wanted equality constraint: $f \; a \; \sim \; T \; x \; y \; z$,
- for a certain class C, the class constraint $C \; f$ is given or wanted,
- and exactly one instance of C for one of the types $\Lambda q \; . \; T \; q \; y \; z$, $\Lambda q \; . \; T \; x \; q \; z$ or $\Lambda q \; . \; T \; x \; y \; q$

then we may conclude that $f \; \sim \; \Lambda \; ... \; . \; T \; x \; y \; z$ and thus $a \; \sim$ ... (where ... is determined by the alternative chosen in the 3rd condition).

This may sound demanding, but in many cases all three will be true. If the first two conditions hold, but the third does not, constraint solving will fail due to a missing instance of C for T anyway.

$$\begin{aligned}
&\text{decompose}(\mathcal{Q} \wedge C\ (\Lambda\overline{y_i}.T\overline{y}), f\ \overline{a} \sim T\overline{x}, C\ f) \\
&\qquad = (f \sim \Lambda x_i.T\overline{x}) \wedge \overline{(a \sim x_i)} \\
&\text{decompose}(\mathcal{Q}, f\ \overline{a} \sim g\ \overline{b}, C\ f \wedge C\ g) \\
&\qquad = (f \sim g) \wedge \overline{(a \sim b)}
\end{aligned}$$

$$\frac{\text{decompose}(\mathcal{Q}, Q_1, Q_2 \wedge Q_3) = Q_4}{\mathcal{Q} \vdash \langle \bar{\alpha}, \varphi, Q_g \wedge Q_3, Q_w \wedge Q_1 \wedge Q_2 \rangle \hookrightarrow \langle \bar{\alpha}, \varphi, Q_g \wedge Q_3, Q_w \wedge Q_2 \wedge Q_4 \rangle}$$

Figure 20. The applied-concrete and applied-applied type checking rules.

A situation where the first condition holds, but no C exists for the second and the third conditions should be rare, as we argued in Section 2.2: this would imply that an applied type variable was used, but neither the applied type variable itself, nor its arguments occur anywhere else (for example in a class constraint). In this situation $f\ \ a$ could be replaced by a fresh, single type variable, which would make the constraint solvable and the code would receive a more general type.

7.3.2 Applied-applied

For equality constraints of the following form:

$$f\ a\ \sim\ g\ b$$

we use a similar rule as in Section 7.3.1: if we have a class constraint that applies to both f and g, then we may split the constraint into $f\ \sim\ g$ and $a\ \sim\ b$.

When we can not find any constraint on both f and g, then the constraint has to stay unsolved. It could be that a different equality constraint can find a substitution for f or g and the constraint will be solved later. If that does not happen, then this constraint should be reported to the user as an error.

There is one exception to this rule:

$$f\ a\ \sim f\ b$$

will still be decomposed automatically, resulting only in $a\ \sim\ b$.

7.3.3 Type rules

We can express the typing rules in the previous two sections formally as in Figure 20. Using the notation from In (Vytiniotis et al. 2011), $\hookrightarrow$ is a typing judgment with an input tuple $\langle \bar{\alpha}, \varphi, Q_g, Q_w \rangle$ and an output tuple $\langle \bar{\alpha}', \varphi', Q_g', Q_w' \rangle$. Here, $\mathcal{Q}$ is the top- level environment (containing, for example, all defined class instances), Q_w are the wanted constraints and Q_g are the given constraints. $\bar{\alpha}$ is a set of touchable variables (the variables which may be substituted) and φ is a set of substitutions. The $\hookrightarrow$ judgment is applied until a fixed-point is found. In (Vytiniotis et al. 2011) a number of cases for $\hookrightarrow$ are described, Figure 20 adds a new case to deal with type-level lambdas in instances. We have also removed a case which is not explicitly given in (Vytiniotis et al. 2011), namely the rule that would automatically decompose $f\ \ a\ \ \sim\ \ g\ \ b$ into $f\ \ \sim\ \ g$ and $a\ \sim\ b$.

The first part of the decompose function in Figure 20 checks the top-level environment for an instance of the class C for the type f, then for an applied-concrete equality constraint and a constraint $C\ \ f$, which may have come from either the given or wanted constraints. The equality constraint is then decomposed according to the type-level lambda used by the instance. The second part of the decompose function checks for an applied-applied equality constraint and a type class that applies to both. Then the equality constraint is split. Here $C\ f$ and $C\ g$ may also come from both the wanted and the given constraints.

7.4 Termination

The goal of the type checker in Haskell is to judge whether programs are well-behaved within a finite amount of time. It is not a problem if programs are rejected when they can not be determined to be well-behaved in a certain amount of time, but it should be impossible to have a non well-behaved program accepted by the type checker.

In particular, this means that it is important that termination of the constraint solver can still be guaranteed.

To any type, we can assign a *depth* as follows:

- The depth of a single type variable or a nullary type constructor is 0.
- The depth of an applied type $t\ \ a\ \ b\ \ c...$ is:

$$1 + \max(\text{depth}(t), \text{depth}(a), \text{depth}(b), \text{depth}(c), \dots)$$

When an applied equality constraint is solved according to one of the two rules we introduced the result is a number of new equality constraints. These can again be applied equality constraints. For example:

$$\begin{aligned}
&[[a]]\ \sim\ f\ (g\ x) \\
&[a]\ \sim\ g\ x, []\ \sim\ f \\
&a\ \sim\ x, []\ \sim\ g, []\ \sim\ f
\end{aligned}$$

However, the newly introduced equality constraints will always have a strictly lower depth than the equality constraint that was solved, because solving a constraint can not introduce a deeper nesting level. This implies that infinite loops in equality constraints are impossible: an equality constraint can only introduce a finite number of equality constraints of lower depth.

7.5 Implementation

The described changes were implemented as a patch for GHC. The patch works with the development version 7.7 and can be found on `https://github.com/xnyhps/ghc/tree/TypeLambdaClasses`.

8. Results

The code shown in Section 1 now works: *fmap* changes the first argument of the data type. As can be seen from this example, no extra type annotations are necessary and the *Functor* class used is unchanged. This means the solution satisfies the two conditions in Section 3.

In Figure 21 we show an example of a *Monad* instance, again defined on the first argument of a data type with three arguments. Additionally a *MaybeT* monad transformer is created where the same type-level lambda is used. From the result it can be seen that the correct instance was found.

9. Limitations with other GHC features

To become part of GHC, our changes should not only be consistent with the Haskell 2010 specification (Marlow 2010), but we should also make sure it works correctly with other GHC extensions, or, if that is impossible, document why combining those extensions leads to problems and add warnings to the compiler when users try to use them at the same time.

```
import Control.Monad.Trans.Class
import Control.Monad.Trans.Maybe

data T x y z = T x

instance Monad (Λx . T x y z) where
  return x = T x
  (>>=) (T x) g = g x

bar :: T String Char Int
bar = return "bar"

foo :: MaybeT (Λx . T x Char Int) String
foo = lift bar
```

```
*Main> runMaybeT foo
T (Just "bar")
```

Figure 21. An example of a $Monad$ instance using a type-level lambda and a $MaybeT$ monad transformer using that same type-level lambda.

```
class C f a b where
  func :: f a b

instance C (Λx y . (x, y)) Int Char where
  func = (1, 'a')

instance C (Λx y . (y, x)) Char Int where
  func = (2, 'b')
```

Figure 22. Despite the instance heads appearing very differently, both *func*s have the same type.

9.1 MultiParamTypeClasses

The `MultiParamTypeClasses` GHC extension (The GHC Team 2013) allows type classes to be specified with multiple type parameters.

Combining this with type-level lambda instances can create new ambiguities, as can be seen in Figure 22.

Without β-evaluating the instance's type-level lambdas, it is not clear that these instances overlap. However, *foo* in both instances has the type $(Int, Char)$.

Ambiguity is not necessarily a problem: GHC does not prohibit two instances to exist which can be ambiguous for some type, only when a class is resolved and multiple instances match an error is raised.

However, it will also complicate the solver's strategy. The solver currently looks for instances for every possible lambda with the required number of arguments for a single type. When using multi-parameter type classes, it meeds to look for every possible lambda abstraction for each of them. This means that increasing the number of parameters can lead to an exponential increase in the number of instances that need to be considered.

Instead of failing on ambiguity only when it occurs, another option would be to apply the following restriction to multiparameter type lambda instances: different instances with the same type constructor on the lambdas body must use the same type-level lambda. This would forbid the example in Figure 22, because while both use the type constructor $(,)$, the order in which they take their arguments is flipped. The advantage of this restriction is that the type-level lambda for every parameter can be found independently, avoiding the exponential increase in cases.

9.2 PolyKinds

Combining the `PolyKinds` extension (The GHC Team 2013) with type-level lambda instances currently has some implementation problems. However, with some more work it should be possible to eliminate those problems.

When `PolyKinds` is enabled, type constructors take implicit kind arguments for all type variables which do not have a fixed kind.

For example, the datatype T:

```
data T x y z = T x
```

Would no longer have type $T :: \forall\ y\ z\ .\ x \rightarrow T\ x\ y\ z$, but instead:

$$\begin{aligned} T :: \forall\ & (k :: \Box) \\ & (l :: \Box) \\ & (x :: *) \\ & (y :: k) \\ & (z :: l)\ . \\ & x \rightarrow T\ k\ l\ x\ y\ z \end{aligned}$$

In practice the user will not see these kinds, so they should not be considered when selecting the type-level lambda decomposition to use. This is not yet implemented.

10. Related work

10.1 Guided Higer-Order Unification

As described in Sections 5.3 and 7.2.1, (Neubauer and Thiemann 2002) introduced a similar approach to type-level lambda terms in instance declarations. However, they also introduced a more complicated unification strategy known as Guided Higher-Order Unification (Λ_{GHOU}).

This strategy also tries to keep unification decidable, but does so in a less restrictive way than our approach. To maintain decidability, Λ_{GHOU} applies the restrictions:

- Identity type-level functions are not allowed.
  $\Lambda x\ .\ x$ is forbidden.
- Constant type-level functions are not allowed.
  This forbids, for example $\Lambda x\ .\ Int$.
- Type-level projection functions are not allowed.
  This forbids functions like $\Lambda x\ y\ .\ x$.

The well-formedness restriction we have presented in our solution is more strict, and we will show the differences.

First of all, Λ_{GHOU} allows lambda-abstracted variables to occur multiple times in the lambda's body, our well-formedness restriction forbids this and only allows those variables to occur exactly once. So these are valid Λ_{GHOU} types, but non-well-formed:

$$\Lambda x\ .\ T\ x\ x\ y$$
$$\Lambda x\ .\ (x, x)$$

Secondly, Λ_{GHOU} also allows lambda-abstracted variables to occur arbitrarily "deep" within the lambda's body, in other words, nested in (an) extra type constructor(s).

Thus, these are also valid Λ_{GHOU}, but non-well-formed:

$$\Lambda x\ .\ T\ [x]\ y\ z$$
$$\Lambda x\ .\ [(x, Int)]$$

As we argued in Section 7.2.1, we believe the extra limitations we impose on type-level lambdas in type classes will not be a

```
trait Monad[M[_]] {
  def point[A](a: A): M[A]
  def bind[A, B](m: M[A])(f: A => M[B]): M[B]
}

class EitherMonad[A]
    extends Monad[({type ?[a] = Either[A, a]})#?] {
  def point[B](b: B): Either[A, B]
  def bind[B, C] (m: Either[A, B])
                 (f: B => Either[A, C]): Either[A, C]
}
```

Figure 23. The *Monad* trait (the Scala equivalent of a type class) in Scala, with an instance for *Either* using a type-level lambda, where the second type variable is used by the *Monad*.

problem for many programmers, as allowing these would create instances that are very different from how type classes are currently used. Programmers can create instances where the order of type variables doesn't matter, which was our initial goal.

10.2 Scala

Scala allows type-level lambdas using the following syntax:

```
({type ?[a] = Either[A, a]})#?
```

This allows class instances to be specified as can be seen in Figure 23.

Scala also allows multiple, different instances for the same type. As mentioned in Section 4.5, this requires a trade-off: in Scala this is done by giving instances names and passing these instances to the function with a class constraint. This is more manual work for the programmer, but it means the implementation for the type checker can be much simpler than the approach described here: the type checker always knows which instance to use before type checking starts, it doesn't need to find the instance based on the types in the constraints.

11. Conclusion

We have described a problem caused by the inflexibility of the combination of type classes and type constructors with multiple arguments. We have presented a number of potential solutions to this problem that are currently supported by GHC and explained how none of them satisfy the desired properties we've specified.

After a brief overview of type checking and constraint solving in Haskell, the concept of type-level lambdas in Haskell was introduced, with a number of restrictions to avoid undecidability in the type checker. A number of changes were proposed to allow the constraint solver to deal with type-level lambdas in type class instances. We have demonstrated that this solves the described problem and has the desired properties we specified, while having only a small impact on existing Haskell code.

References

T. P. Alkemade. UnitTyped. `https://hackage.haskell.org/package/unittyped`, 2012. [Online; accessed 9-April-2014].

C. Elliott. TypeCompose. `https://hackage.haskell.org/package/TypeCompose`, 2007-2013. [Online; accessed 10-December-2013].

A. Gill. MTL. `https://hackage.haskell.org/package/mtl`, 2006-2012. [Online; accessed 10-December-2013].

A. Gill and R. Paterson. Transformers, 2009-2012. [Online; accessed 10-December-2013].

G. Gonzalez. mmorph. `https://hackage.haskell.org/package/mmorph`, 2013-2014. [Online; accessed 20-February-2014].

C. Hall, K. Hammond, S. P. Jones, and P. Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems*, 18:241–256, 1996.

G. P. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257 – 267, 1973. ISSN 0019-9958.

E. A. Kmett. Bifunctors. `https://hackage.haskell.org/package/bifunctors`, 2011-2013. [Online; accessed 10-December-2013].

S. Marlow. Haskell 2010 language report, 2010.

M. Neubauer and P. Thiemann. Type classes with more higher-order polymorphism. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 179–190, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

J. Peterson and M. P. Jones. Implementing type classes. In R. Cartwright, editor, *PLDI*, pages 227–236. ACM, 1993. ISBN 0-89791-598-4.

The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide*, April 2013.

D. Vytiniotis, S. Peyton jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, sep 2011. ISSN 0956-7968.