Type Class Instances for Type-Level Lambdas in Haskell

Thijs Alkemade and Johan Jeuring

Technical Report UU-CS-2015-014
October 2015

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Type Class Instances for Type-Level Lambdas in Haskell

Thijs Alkemade and Johan Jeuring

Utrecht University

Abstract. Haskell 2010 lacks flexibility in 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 allows to do this in a satisfactory way.
To flexibly create type-class instances 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

The first version of the `unittyped` package [1] used a datatype similar to:

```
data Value v u d = Value v
```

A *Value v u d* contains an object of type v and is tagged with phantom types u and d. The type u represents the physical unit of the value (meters, miles, seconds, etc.) and d the dimension of that unit (length, time, etc.). Using type classes, 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 first release, a feature request asked for a *Functor* instance for *Values*. *Functor* is a type class given by:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Fig. 1. The definition of *Functor*.

The only possible instance that the datatype would allow would give *fmap* the type:

```
fmap :: (a → b) → Value v u a → Value v u b
```

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

```
fmap :: (a → b) → Value a u d → 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:

```
data Value u d v = Value v
```

A type class in Haskell is a set of polymorphic functions that can only be used on types that have instances for that class [5]. 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 arrow kinds 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 [10]. For example, *Eq* has a type variable of kind $*$, as the type variable a in the class head does not receive any arguments:

```
class Eq a where
  (≡) :: a → a → Bool
```

The *Functor* class has a type variable of kind $* \to *$, as f is used with one argument (as a and b are of kind $*$). See Fig. 1.

One example of a type class using a type of kind $* \to * \to *$ is *Category* (Fig. 2).

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

Fig. 2. The definition of *Category*.

The order of arguments for a function is usually selected based on what is the most convenient. For example, similar functions can be given similar orderings of arguments: the functions in `Data.Map` that receive a single *Map* argument receive the map as their last parameter. When a partially applied function is needed and the missing parameters are not the last arguments, then functions like *flip* and λ-abstractions can be used to rearrange the arguments to obtain any desired ordering of parameters.

Choosing the order of the type arguments of a type constructor may appear to be similar. The exact order is important when considering which higher-kinded types can be formed, but an equivalent of *flip* does not exist. For example, a type constructor with three arguments allows only three types of higher kinds to be formed:

```
Value       :: * → * → * → *
Value v     :: * → * → *
Value v u   :: * → *
Value v u d :: *
```

The higher-kinded type *Value* $\cdot$ *u d*, of kind $* \to *$, where a type argument is substituted at the position of the $\cdot$, cannot be constructed in this way. This also means that, if a one class requires them to be in a certain order, but another class requires a different order, then it is impossible to give both instances at the same time.

In this paper we show how we can extend Haskell with a restricted form of type-level lambdas in class instance heads, so that we for example can write the following:

```
data Value v u d = Value v
instance Functor (Λv . Value v u d) where
```

```
fmap f (Value v) = Value (f v)
```

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

In other words, the goal of this paper is to make the order of the arguments of a type constructor flexible. In particular, they should have no effect on how instances of higher-order classes can be defined. We require our solution to satisfy 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.

The rest of this paper is organized as follows: Section 2 explains potential solutions to this problem using existing techniques and approaches, and the problems with these solutions. Section 3 formalizes the notion of type-level lambdas. Section 4 gives a general background about type checking and constraint solving in Haskell, and Section 5 gives the changes necessary to support type-level lambdas in GHC. Section 6 shows what is possible with these changes and Section 7 lists some potential problems with other GHC extensions.

2 Using existing concepts

This section discusses how existing concepts in Haskell might be used to solve the problem described in the previous section. None of our attempts solves the problem satisfactorily.

Currently, the simplest solution to obtain the correct instances for a type is 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 *Value v u d* later needs to have *Functor* instance that works on v, that would mean replacing every *Value v u d* with *Value u d v*. In a large codebase, this could be a significant amount of work.

2.1 Type Synonym Instances

The GHC extension `TypeSynonymInstances` [11] may appear to be 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.

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. For example, we can create a new type synonym for *Value* which specifies the order of the type variables for its *Functor* instance.

```
data Value v u d = Value v
type ValueFunctor u d v = Value v u d
instance Functor (ValueFunctor u d) where
  fmap :: (a → b) → Value a u d → Value b u d
```

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 cannot be partially applied to form a type of kind $* \to *$ and be supplied to *Functor*. We will look further into this restriction in Section 3.1.

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

```
data Value v u d = Value v
type ValueFunctor y x = Value v y
```

But this implies we are back at supporting only the limited set of higher-kinded types given in Section 1. This solution does not meet our main goal.

2.2 More type classes

Another 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 is added. The `Bifunctors` package [7] 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. For example, we can create *Functor*-like classes using the second and third variable with:

```
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
```

The advantage of this solution is that instances for *Functor2* and *Functor3* do not overlap. For every variable of a type constructor it is possible to indicate whether or not 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*:

```
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)
```

Where the main goal of type classes is to avoid code duplication, this solution adds code duplication. Another disadvantage of 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 $* \to * \to *$ (Fig. 2). Every possible pair would require a separate class, *Category_2_3*, *Category_1_3*, *Category_2_1*, etc.

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

2.3 Newtype wrappers

The `TypeCompose` package [3] contains the newtype definition:

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

Flip can be viewed as a type-level variant of *flip*: the last two type arguments of the *Flip* type constructor are the last two type arguments of the wrapped type, but swapped. This also makes it possible to write instances where the last two variables are swapped:

```
data Value u v d = Value v
instance Functor (Flip (Value u) d) where
  fmap :: (v → v′)
```

```
    →  Flip (Value u) d v
    →  Flip (Value u) d v'
fmap f (Flip {unFlip = Value v})
       = Flip {unFlip = Value (f v)}
```

It is not only possible to flip the last two arguments, but *Flip* can be generalized to every reordering of type variables:

```
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}
```

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

A disadvantage of this solution is that every value to which we want to apply a method from the class instance for its type needs to be wrapped with *Flip*, and unwrapped with an *unFlip* call. For example, we could apply a function to a wrapped type as:

```
fmap (+1) (Value 42) ⇒ unFlip (fmap (+1) (Flip (Value 42)))
```

This solution meets our main goal, but the extra boilerplate code necessary to wrap and unwrap datatypes before and after applying type class methods means that it does not satisfy the second condition.

2.4 Associated type families

The reason why the *Functor* class needs instances with a variable $f :: * \to *$ is to make it possible to construct $f\ a$ and $f\ b$. 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 only possible to declare a *Functor* instance using the type parameter in the last position.

With the `TypeFamilies` language extension of GHC [2] 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:

```
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 Value v u d = Value v
instance Functor (Value v u d) where
  type FunctorApp (Value v u d) v' = Value v' u d
  fmap :: (a → b) → FunctorApp (Value v u d) a
                  → FunctorApp (Value v u d) b
  fmap f (Value v) = Value (f x)
```

Note that *Functor* no longer receives a type of kind $* \to *$ but of kind $*$, because it doesn't 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 is 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.

This solution has a problem due to the way type families currently work: to call *fmap* (+1) on *Just* 1, the type family equality constraint *FunctorApp f a* $\sim$ *Maybe Int* needs 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 define *FunctorApp T v* = *Maybe x*, so the equality constraint cannot be solved. This makes type families unusable for our goal.

2.5 Conclusion

None of the mentioned solutions satisfies the main goal and the conditions in Section 1.

A number of the approaches allow multiple instances per type constructor for a given type class. For example, some allow defining *Functor* instances (or a new instance intended to look like *Functor*, such as *Functor2*) for both the last and the second to last type variable. Although this may be useful, it cannot satisfy both conditions at the same time: without either type annotations or boilerplate code, the compiler is unable to determine which instance to use for an *fmap* call, for example:

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)

We shall therefore consider multiple instances for the same type overlapping, even when they use a different ordering of type variables.

3 Type-level lambdas

In the previous sections we have informally used *Λ* to denote a type-level lambda function in class instance heads. In this section we will give a precise definition of a restricted form of type-level lambdas, and we will investigate the issues it can create with type checking.

3.1 Undecidability

Let us examine more closely why the solution in Section 2.1 is rejected by the compiler. Type synonyms can have zero or more arguments. However, contrary to type families they cannot do any case distinction on those variables. We can consider type synonyms polymorphic "functions" on types.

To be able to use a type class function, the compiler needs to be able to determine for each instance of that class whether the type inferred for the function can be made equal to the type of the instance, possibly by substituting some free type variables. If a single instance matches then the class constraint has been resolved and the correct implementation of the class function can be looked up.

Determining whether a substitution of free variables exists which makes a partially applied type synonym equal to a type comes down to determining whether two lambda expressions are equivalent. This problem is known as unification. Specifically, it is higher-order unification: free variables may be replaced by new lambda abstractions.

Higher-order unification is undecidable in general [6]. Haskell therefore uses the rule that type synonyms must be fully applied before they may be used as a type. That is why the example in Section 2.1 is rejected: it is using a partially applied type synonym where a type is expected.

3.2 Adding Type-level Lambdas

To define an instance of *Functor* where *fmap* has type $(a \to b) \to \mathit{Value}\ a\ u\ d \to \mathit{Value}\ b\ u\ d$, the instance head would need to have a type τ such that $\tau\ a$ is equal to $\mathit{Value}\ a\ u\ d$ and $\tau\ b$ is equal to $\mathit{Value}\ b\ u\ d$. Type synonyms can not be used in instance heads, but even if they could, it would be more convenient to have a 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*: $\Lambda v\ .\ \mathit{Value}\ v\ u\ d$. This is a type where v is bound by the lambda, and u and d are free.

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

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

The kind checking rule for type-level lambdas is given by:

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

here $\mathcal{Q}$ 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.

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 Section 8.2. Although type-level lambdas do not exist in Haskell itself, they do occur in Core, the typed internal representation of GHC.

Note that $\forall\ u\ .\ \mathit{Value}\ v\ u\ d$ and $\Lambda u\ .\ \mathit{Value}\ v\ u\ d$ do not mean the same thing. $\forall\ u\ .\ \mathit{Value}\ v\ u\ d$ has the same kind as $\mathit{Value}\ v\ u\ d$, but $\Lambda u\ .\ \mathit{Value}\ v\ u\ d$ has kind $l \to k$, with $u :: l$ and $\mathit{Value}\ v\ u\ d :: k$.

3.3 Decidable unification of type-level lambda terms

We can view a type-level lambda as an anonymous type synonym, just like a value-level lambda function is an anonymous function. Unification of type-level lambdas will therefore be undecidable in general. We can use multiple solutions for this:

1. Use the same restriction as for type synonyms: a type-level lambda needs to be fully applied before it may be used as a type, with an exception for instance heads.
   A disadvantage of this solution is that for example monad transformers, which take a type variable with a *Monad* constraint, cannot be specified for monads that use type-level lambdas. For example, $\mathit{MaybeT}\ (\Lambda u\ .\ \mathit{Value}\ v\ u\ d)\ a$ would be forbidden.

   $$\begin{aligned}&\textbf{newtype}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MaybeT}\ m\ a\\ &\quad = \mathit{MaybeT}\ \{\mathit{runMaybeT} :: m\ (\mathit{Maybe}\ a)\}\end{aligned}$$

2. We can try to restrict the unification problems to a subset that is decidable.
   First-order unification (unification where no new λ-abstractions may be introduced) is decidable, but not sufficient for our goal: it would be unable to unify $f\ a$ with anything.
   Higher-order matching is also decidable. Matching is unification where one of the two arguments contains no free variables. However, this is also not useful in Haskell: the type in a type class instances may contain free variables.
3. Use Guided Higher-Order Unification (Λ_{GHOU}) as proposed by [9]. See Section 8.1.
4. The above three solutions are either too restrictive or very complicated. Instead, we choose to allow unapplied type-level lambda functions. Except for in instance heads (see Section 5.2), type-level lambdas are not unified: they must be α-equivalent or type checking will fail. This implies that they may be used in monad transformers, which is impossible in the first option, but they must be used consistently.

4 Type checking Haskell

This section briefly describes type checking and constraint solving for Haskell as used in GHC [12].

Type checking is split into two phases: first types are inferred, for which constraints are generated, and then these constraints are solved. Solving these constraints gives a set of substitutions and a (possibly empty) set of unsolved constraints.

4.1 Example

Consider the following program:

$$f\ x = x + 5$$

The type inferencer starts with giving f a function type, because it takes an argument. x gets assigned the argument's type.

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

By looking up the *Num* class, the type inferencer determines 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 introduces the constraints $\alpha \sim a$, $b \sim a$ and $\beta \sim a$. So the type inferencer finishes with the set of constraints $(\alpha \sim a, b \sim a, \beta \sim a, Num\ a, Num\ b)$.

The constraint solver turns these equality constraints into substitutions, as they are all 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 $(Num\ a) \Rightarrow a \to a$.

4.2 Generating constraints

The important constraints to consider are:

1. **Class constraints:** a class followed by zero or more types:
   $D\ x$
   We only look at classes that use exactly one variable, see also Section 7.1.
2. **Equality constraints:** constraints requiring two types to be equal:
   $a \sim b$
   We consider different types of equality constraints:

(a) **Impossible:** Equality constraints that contain different concrete types on both sides cannot 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 that contain a single type variable on one side:

$a \sim T$

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

$F\ a \sim T$

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

$f\ a \sim g\ b$

$f\ a \sim [Int]$

4.3 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 over 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, it rejects equality constraints where the same variable occurs on the lhs and the rhs and the lhs and the rhs are not equal, as these would require an infinite type (the "occurs check").

   If a type family is used within another type family, then these are split into two separate type family constraints.

   We use `[W]` to denote wanted constraints (generated during type checking) and `[G]` to denote given constraints (given by the user by supplying a type signature). Some examples of the canonicalization step are:

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

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

   This is allowed because f and g have to be (partially applied) type constructors, not type synonyms or type families. As described in Section 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 two canonical constraints together. For example, a simple equality constraint and another constraint will apply the equality constraint as a substitution to the other constraint. Having two identical constraints implies 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:

   $\{\texttt{[W]}\ Num\ a, \texttt{[W]}\ Num\ a\} \rightarrow \{\texttt{[W]}\ Num\ a\}$

   $\{\texttt{[W]}\ a \sim T, \texttt{[W]}\ Num\ a\}$
   $\rightarrow \{\texttt{[W]}\ a \sim T, \texttt{[W]}\ Num\ T\}$

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

3. **Simplification:** The simplifier also looks at two canonical constraints, but specifically pairs of constraints where one of them is given and the other is wanted.
   Obviously, a given constraint and a wanted constraint that are identical implies that the wanted constraint can be deleted. Given simple equality constraints are be used as substitutions on wanted constraints.
   Here are some examples:
   $\{\texttt{[W]}\ \mathit{Functor}\ f, \texttt{[G]}\ \mathit{Functor}\ f\} \rightarrow \{\texttt{[G]}\ \mathit{Functor}\ f\}$
   $\{\texttt{[W]}\ \mathit{Functor}\ f, \texttt{[G]}\ f\ \sim\ g\}$
   $\rightarrow \{\texttt{[W]}\ \mathit{Functor}\ g, \texttt{[G]}\ f\ \sim\ g\}$
4. **Top-level interaction:** The top-level interaction stage is the stage where the class instances, type family instances and equality constraints given in the code are used to solve wanted constraints. 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, if the usual *Functor* [ ] instance is in scope, then we may eliminate all *Functor* [ ] constraint:
   $\{\texttt{[W]}\ \mathit{Functor}\ [\,]\} \rightarrow \{\,\}$
   Suppose we have a type family:
   type family $F\ x\ \ ::\ *$
   type instance $F\ \mathit{Int} = [\mathit{Int}]$
   Then the top-level interaction step produces the following constraint:
   $\{\texttt{[W]}\ a\ \sim\ F\ \mathit{Int}\} \rightarrow \{\texttt{[W]}\ a\ \sim\ [\mathit{Int}]\}$

5 Adding Type-Level Lambdas to GHC

We have started 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.

The changes to GHC 7.7 consist of three parts: the parser is modified to allow the notation `/\` for type-level lambdas, the internal representation of types is modified to support Λ and the constraint solving is adapted to take the possibility of type-level lambdas in class instance heads into account.

`/\` is currently valid syntax for a term-level operator. Because our extension is type-level syntax this does not cause problems. With the GHC extension `TypeOperators` [11], `/\` can be used as a valid type operator. We assume that because using it requires a relatively uncommon extension, there will not be many problems with existing code already using this syntax.

5.1 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 the `KindSignatures` extension [11] is enabled. For example:

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

5.2 Evaluation of type-level lambdas

We evaluate type-level lambdas greedily during type checking: when a type- level lambda is encountered that is applied to an argument, the substitution is carried out immediately. We show that this cannot 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 implies that we cannot write non-terminating combinators from the untyped lambda calculus, such as Ω:

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

A value-level ω combinator cannot be type checked as its type fails the occurs check: suppose $\omega :: \tau \to \sigma$, then $\sigma = \tau\ \tau$, which implies $\tau = \tau \to \sigma$. This equation cannot hold for types. Thus Ω also fails to type check. In GHC, trying to define ω would cause a "occurs check" error. At the type-level, defining ω (by either a type-level lambda or a type synonym) is also rejected, as the occurs check for its kind would fail.

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 cannot refer to themselves. It would be possible if the Y combinator were available, 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. Haskell currently supports two different forms of type-level functions:

- Type synonyms
- Type families

Type synonyms are not allowed to be recursive: trying to create a (mutual) recursive type synonym will give an error "Cycle in type synonym declarations".

Type families can be (mutually) recursive, but only when the `UndecidableInstances` [11] extension is enabled. When this extension is not enabled, recursion is forbidden because it will imply that an 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` causes GHC to lift many of its restrictions that guarantee termination. With this flag on, it is already possible to create infinite loops in the type checker, using only type families. Adding type-level lambdas does not cause code that was previously terminating to be come non-terminating.

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 only allow well-formed type-level lambdas as instance heads.

Definition 1 *A* well-formed *type-level lambda function is an expression that can be constructed using the following grammar:*

T	*(type constructor)*
a	*(type variable)*
$as \coloneqq a_1, \ldots, a_n$	*(1 or more)*
$ts \coloneqq t_0, \ldots, t_m$	*(0 or more)*
$t \coloneqq a \mid T\ ts$	*(monotype)*
$L \coloneqq \Lambda as.T\ t\ s$	*(type lambda)*

under the extra condition that every variable that is bound by a `/\` *must occur exactly once as an argument to the inner type constructor. In other words, a type-level lambda 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.*

Some examples of well-formed types:

$\Lambda x\ .\ \mathit{Value}\ v\ u\ d$
$\Lambda x\ .\ [x]$
$\Lambda x\ .\Lambda y\ .\Lambda z\ .\ \mathit{Value}\ z\ x\ y$

and some not well-formed types:

$\Lambda x\ .\ \mathit{Value}\ [x]\ y\ z$
$\Lambda x\ .\ \mathit{Value}\ v\ x\ z$
$\Lambda x\ .\ [\mathit{Int}]$

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

For most instances given by programmers the well-formedness restriction should not cause problems: it allows reordering of type variables (which is what we want 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.

We give some examples of instances that cannot be written because they use non well-formed type-level lambdas.

- Definition 1 states that the body of a type-level lambda starts with either another type-level lambda, or a type constructor. In particular, it does not start with a type variable, including the lambda-bound type variables. This implies for example that an instance for $\Lambda x\ .\ x\ \mathit{Int}$ cannot be specified. However, this type would have kind $(* \to *) \to *$ (which 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 occurs as a direct argument to the inner type constructor. It may not be inside another type constructor in the argument. This implies 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 type variable occurs 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:

  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**
  $\mathit{fmap}\ f\ x = \mathit{fmap}\ (\mathit{fmap}\ f)\ x$

  However, this does not work, as both instances' heads are not well-formed. Although these instances 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 $\mathit{fmap}\ f$ 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 cannot resolve whether the call to $\mathit{fmap}\ f$ is meant to apply on

the first functor alone, or on the composition. *fmap* (*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 [4] to obtain identity functors and functor composition.

5.3 Constraint solving

We can now express type class instances with type-level lambda instances, but to use them the constraint solver needs 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 4.3) is no longer allowed. Suppose we have the following datatype, and in the code *fmap* is applied to it:

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

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

- $f\ \sim \Lambda b\ .\ T\ b\ y\ z$ and $a\ \sim\ Int$
- $f\ \sim \Lambda b\ .\ Value\ v\ b\ z$ and $a\ \sim\ ()$
- $f\ \sim \Lambda b\ .\ Value\ v\ y\ b$ and $a\ \sim\ Char$

The constraint cannot be split, but it has to be solved as a whole. First, we give the requirements for how these constraints can be split (an applied type variable on one side, a concrete type on the other) and second we explain how to deal with the case where both sides consist of an applied type variable.

Applied-concrete Suppose a constraint $f\ a\ \sim\ Value\ v\ u\ d$ is encountered: an applied type variable on one side, with a concrete type on the other side.

As mentioned in Section 2.5, satisfying all the conditions from Section 1 at once can only be done when every type 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 constraints are given:

- The wanted equality constraint: $f\ a\ \sim\ Value\ v\ u\ d$,
- 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 $\Lambda q\ .\ Value\ q\ y\ z$, $\Lambda q\ .\ Value\ v\ q\ z$ or $\Lambda q\ .\ Value\ v\ y\ q$

then we may conclude that $f\ \sim \Lambda ?\ .\ Value\ v\ u\ d$ and thus $a\ \sim\ ?$ (where ? is determined by the alternative chosen in the third condition). This may sound restrictive, but in many cases all three will be true. If the first two conditions hold, but the third does not, constraint solving fails due to a missing instance of C for T anyway.

A situation where the first condition holds, but no C exists for the second and the third conditions should be rare, however, not impossible. When $f\ a$ is given in a type signature this almost certainly means there is also a constraint on f: without a constraint, it is impossible to obtain a value of type a or $f\ b$ from a value of type $f\ a$, so the function can only produce values of type $f\ a$. This implies the function has a more general type: by replacing $f\ a$ by a new type variable.

For example, it is possible to write:

$$const :: f\ a \to b \to f\ a$$
$$const\ x\ \_ = x$$

This function can only be applied to datatypes with at least one argument, but it does not use that information. It could be replaced by:

$$const :: c \to b \to c$$
$$const\ x\ \_ = x$$

We do not expect this restriction to have impact on existing Haskell code.

Applied-applied For an equality constraint of the following form:

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

we use a similar rule as in Section 5.3: 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 cannot find any constraint on both f and g, then the constraint stays unsolved. Maybe 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 is reported to the user as an error. There is one exception to this rule:

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

is decomposed automatically, resulting only in $a\ \sim\ b$.

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

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

Fig. 3. The applied-concrete and applied-applied type checking rules.

Type rules We express the typing rules from the previous two sections formally in Fig. 3. Using the notation from In [12], $\hookrightarrow$ 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 [12] a number of cases for $\hookrightarrow$ are described, Fig. 3 adds a new case to deal with type-level lambdas in instances. We have removed a case that is not explicitly given in [12], namely the rule that automatically decomposes $f\ a\ \sim\ g\ b$ into $f\ \sim\ g$ and $a\ \sim\ b$.

The first part of the decompose function in Fig. 3 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.

5.4 Termination

The goal of the type checker in Haskell is to judge whether programs are well-behaved within a finite number of steps. It is not a problem if programs are rejected when they cannot be determined to be well-behaved, but the type checker should not accept a not well-behaved program. In particular, this implies that it is important that the constraint solver terminates. 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:

$$[[a]] \sim f\ (g\ x)$$
$$[a] \sim g\ x, [] \sim f$$
$$a \sim x, [] \sim g, [] \sim f$$

However, the newly introduced equality constraints always have a strictly lower depth than the equality constraint that is solved, because solving a constraint cannot introduce a deeper or equally deep 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.

5.5 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`.

6 Results

The code shown in Section 1 now works: *fmap* changes the values of the first type argument of the datatype. This example shows that no extra type annotations are necessary, and the *Functor* class used is unchanged. This implies that the solution satisfies the two conditions from Section 1.

Here is an an example of a *Monad* instance, again defined on the first type argument of a datatype with three arguments. Additionally a *MaybeT* monad transformer is used where the same type-level lambda is used. The returned result shows that the correct instance is found.

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

data Value v u d = Value v

instance Monad (Λv . Value v u d) where
  return v = Value v
```

```
  (>>=) (Value v) g = g v
bar :: Value String Char Int
bar = return "bar"
foo :: MaybeT (Λv . Value v Char Int) String
foo = lift bar

---

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

7 Compatibility with other GHC features

To become part of GHC, our changes should not only be consistent with the Haskell 2010 specification [8], but we should also make sure they work 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.

7.1 MultiParamTypeClasses

The `MultiParamTypeClasses` GHC extension [11] allows type classes to be specified with multiple type parameters. Combining multiple type parameters with type-level lambda instances can create new ambiguities:

```
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')
```

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

Ambiguity is not necessarily a problem: GHC does not prohibit two instances to exist that can be ambiguous for some type, only when a class is resolved and multiple instances match an error is raised. However, ambiguity complicates 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 multiparameter type classes, it needs 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 is to apply the following restriction to multiparameter type-level lambda instances: different instances with the same type constructor on the lambdas body must use the same type-level lambda. This forbids the given example, 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.

7.2 PolyKinds

Combining the `PolyKinds` extension [11] with type-level lambda instances currently has some implementation problems. However, we expect that with some more work it is 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 constructor *Value* of the datatype:

data *Value* $v\ u\ d = Value\ v$

does no longer have type $Value :: \forall\ u\ d\ .\ v \to Value\ v\ u\ d$, but instead:

$$\begin{aligned} Value :: \forall\ & (k :: \Box) \\ & (l\ :: \Box) \\ & (v :: *) \\ & (u :: k) \\ & (d :: l)\ . \\ & v \to Value\ k\ l\ v\ u\ d \end{aligned}$$

In practice the user does not see these kinds, so when decomposing an applied equality constraint involving *Value*, these extra kind arguments should not be considered when selecting the type-level lambda to use. However, they currently are, causing unsolved constraints in certain situations.

8 Related work

8.1 Guided Higher-Order Unification

As already mentioned in Sections 3.3 and 5.2, [9] introduce a similar approach to type-level lambdas in instance declarations. They introduce a more involved unification strategy known as Guided Higher-Order Unification (Λ_{GHOU}). This strategy also keeps unification decidable, but in a less restrictive way than our approach. To maintain decidability, Λ_{GHOU} applies the restrictions to solving type-level Λ constraints:

- 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 $\Lambda x\ y\ .\ x$.

Additionally, given an instance:

instance $C\ (\Lambda x_1 \ldots x_n . T\ t_1 \ldots t_m)$ **where**

they apply the following restrictions:

- Each variable that is free in $\Lambda x_1 \ldots x_n . T\ t_1 \ldots t_n$ occurs once.
- Each x_i occurs free in $T\ t_1 \ldots t_n$.
- Each t_i is either one of the x_j, or equal to $g\ y_1\ y_l$, where $g \notin \{x_i\}$ and $\{y_j\} \subseteq \{x_i\}$

The well-formedness restriction we introduce is more strict. We discuss the differences. Firstly, 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 not well-formed:

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

Secondly, Λ_{GHOU} 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 not well-formed:

$\Lambda x\ .\ \mathit{Value}\ [x]\ y\ z$
$\Lambda x\ .\ [(x, \mathit{Int})]$

Λ_{GHOU} applies the same restriction on overlapping instances as our approach: instances using different Λ-abstractions, but the same class and type, overlap. This implies that an instance for $[(x, \mathit{Int})]$ will still overlap all other instances using $[a]$.

As we argued in Section 5.2, we believe the extra limitations we impose on type-level lambdas in type classes will not be a 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.

8.2 Scala

Scala allows type-level lambdas using the following syntax:

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

Here, `{type ?[a] = ... }` introduces a new type alias named `?` with a single argument `a`. It is defined to be equal to `Either[A, a]`. Finally, `#?` projects the `?` type from the type alias.

This allows class instances to be defined as in Fig. 4.

Scala also allows multiple, different instances for the same type. As mentioned in Section 2.5, 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 implies that the implementation of 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 does not need to find the instance based on the types in the constraints.

```
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]
}
```

Fig. 4. 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*.

9 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 satisfies a number of desired properties.

We introduce a restricted form of type-level lambdas in Haskell, which we have added to a development version of GHC. The restrictions are necessary to avoid undecidability in the type

checker. Our solution is simpler than previous attempts, the main task is to adapt the constraint-solver to deal with our form of type-level lambdas. Despite its simplicity, our approach is powerful enough to solve the inflexible type class instances problem in a way that satisfies the desired properties. The solution has virtually no impact on existing Haskell code.

10 Acknowledgments

We would like to thank the anonymous reviewers for their extensive comments.

References

1. Thijs P. Alkemade. UnitTyped. `https://hackage.haskell.org/package/unittyped`, 2012. [Online; accessed 9-April-2014].
2. Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of POPL '05: the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–13. ACM, 2005.
3. Conal Elliott. TypeCompose. `https://hackage.haskell.org/package/TypeCompose`, 2007-2013. [Online; accessed 10-December-2013].
4. Andy Gill and Ross Paterson. Transformers, 2009-2012. [Online; accessed 10-December-2013].
5. Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems*, 18:241–256, 1996.
6. Gerard P. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257 – 267, 1973.
7. Edward A. Kmett. Bifunctors. `https://hackage.haskell.org/package/bifunctors`, 2011-2013. [Online; accessed 10-December-2013].
8. Simon Marlow. Haskell 2010 language report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.
9. Matthias Neubauer and Peter 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.
10. John Peterson and Mark Jones. Implementing Type Classes. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 227–236, New York, NY, USA, 1993. ACM.
11. The GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide. `http://www.haskell.org/ghc/docs/7.6.3/html/users_guide/index.html`, April 2013.
12. Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X) Modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011.