Dialects of Type Computations in Haskell

Type clases, Type Families, and their variaties

APOORV INGLE, University of Iowa, USA

In modern programming languages, it is a well perceived notion, that static types have a two fold advantage (1) It serves as a guiding tool to help programmers write correct code (2) the type checker can help identify code that does not have a correct behavior. An expressive type system can guarantee stronger claims about the written programs. Typeclasses with functional dependencies and type families are two ways of having an expressive type system and performing computations at type level types in Haskell. This paper serves as a primer on the dark arts of typeclasses with functional dependencies and type families along with their generalization, constraint type families.

Additional Key Words and Phrases: typeclass, type family

1 INTRODUCTION

Type classes and their extensions and type families are two powerful extensions to the type system of the Haskell language. Type classes, originally introduced by [Wadler and Blott 1989] was extended to have multiple parameters that represented relations on types. [Jones 1994]'s theory of qualified types gave an account of type classes by introducing predicates or constraints on types at syntax level.

2 PRELIMINARIES

3 TYPECLASSES

Type classes were originally introduced to avoid the problem of implicit code blowup while supporting overloading of operators in languages based on polymorphic lambda calculus. For example, consider built-in or primitive types such as `Int` and `Char`. We want to define a function `inc :: a → a` that when applied to a value of type `Int` returns the next `Int`, similarly, when applied to a value of type `Char` returns the next `Char`. Both these types would have specialized versions of how to return the next value say `incInt :: Int → Int` and `incChar :: Char → Char`. Now, all the functions that use the polymorphic function `inc` would need to resolve the overloaded operator `inc` it to a specific instance during compile time. In a naïve compilation strategy, such a function would need to be generated for each instance of the ground type where it is defined. In this example, we would have to generate two copies, one for `Int` and other for `Char`. While this method works in principle, this is inefficient and leads to a code blowup during compile time.

To solve this problem of code blowup, [Wadler and Blott 1989] suggested the use of dictionaries. In this method, each polymorphic function would be rewritten by the compiler where it would take in an extra implicit dictionary argument that is parameterized by the type to be used at call site. Continuing with the `inc` example its new type would be `inc :: Incable a ⇒ a → a`. To assert that any type τ indeed is defined on the function `inc` we declare a typeclass instance `Incable` τ with `inc` function associated with it. This can be achieved in Haskell using the syntax shown in Figure 1.

Author's address: Apoorv Ingle, University of Iowa, Department of Computer Science, McLean Hall, Iowa City, Iowa, USA.

```
class Incable a where
  inc :: a → a

instance Incable Int where
  inc = incInt

instance Incable Char where
  inc = incChar

inc :: Incable a ⇒ a → a
inc d c = d.inc c
```

Fig. 1. Incable Typeclass and its Instances

One can even have a hierarchy of typeclasses to enable code reuse. From abstract algebra, A `Semigroup` is a collection of elements closed under an associative binary operation and a `Group` is nothing but a `Semigroup` that also has inverses and a unique identity element. To model integers as a group with addition as a binary operation and `0` as the identity element with the inverse operation being just the negative of the integer we can define the instances `Semigroup Int` and `Group Int` as shown in Figure 2. In the dictionary representation of `Group` it would contain the method of its superclass `Semigroup` as well as the methods declared within the class.

```
class Semigroup a where
    (∗)    :: a → a → a

instance Semigroup Int where
    a ∗ b = a + b

instance (Semigroup a, Semigroup)
    ⇒ Semigroup (a, b) where
    (a1, b1) ∗ (a2, b2) = (a1 ∗ a2, b1 ∗ b2)
```

```
class Semigroup a ⇒ Group a where
    ident  :: a
    invert :: a → a

instance Group Int where
    ident   = 0
    invert a = -a

instance (Semigroup a, Semigroup b)
    ⇒ Group (a, b) where
    ident  = (ident, ident)
    invert (a, b) = (invert a, invert b)
```

Fig. 2. A Hierarchy of Typeclasses

In comparison with other languages, as a first approximation, typeclasses look like interfaces from object oriented languages like Java. However there are subtle differences. It would not be possible to define an interface like `Semigroup` in Java that is polymorphic in more than one parameter. This would have to be achieved using generics such as `interface Comparable<T>(T obj) { ... }`. Typeclasses are also different from ML modules as the latter helps achieve data abstraction while the former does not.

3.1 Multiparameter Typeclasses

A natural extension to typeclasses with single parameter would be to have multiple parameters. A multiparameter typeclass with n type parameters would then represent a relation over n types. An example of such a typeclass would be `Add a b c` that represents all the types that support an add `(+)` operation. We would expect to have instances such as `Add Int Int Int`, `Add Int Float Float`, and so on. The instances of typeclass `TEq` would assert that types `a` and `b` that are equal.

```
class Add m n p where
  (+) :: m → n → p

class TEq a b

instance Add Int Float Float where
  (+) a b = addFloat (toFloat a) b

instance TEq Int Int

instance Add Int Int Int
  (+) a b = intAdd a b

instance TEq Char Char
```

Fig. 3. multiparameter Typeclasses

In Figure 3, notice how TEq typeclass has no operations associated with it. In this view, we are no longer using typeclasses to give an implementation of polymorphic functions at specific types. Its sole purpose is to define a predicate (or a relation) on types. We can describe Add and TEq in simple set semantics as, Add = {(*Int*, *Int*, *Int*), (*Int*, *Float*, *Float*)} and TEq = {(*Int*, *Int*), (*Char*, *Char*)}.

Multiparameter typeclasses, while being powerful are problematic in practice. It is possible for a programmer to declare conflicting instances of such a type class. Declaring `Add Int Float Float` and `Add Int Float Int` would be conflicting as the type inference algorithm will no longer be able to resolve the overloaded operator +. The type system has no mechanism to identify such inconsistent instances and disallow them at compile time. The situation gets worse as the type error would be raised at the use of the overloaded operator confusing the programmer about the cause of the issue. However, there may be cases where we do would want to allow such definitions as well. Consider a class `Convert a b` that converts a type a to a type b. All four instances of this class— `Convert Int Int64`, `Convert Int64 Int` `Convert Int Int` and `Convert Int64 Int64`—are valid.

3.2 Functional Dependencies with Examples

Functional dependencies[Jones 2000] is a generalization of multiparameter type classes. It introduces a new syntax where the user can specify a relation between the type parameters while declaring the typeclass. There is no change while declaring the instances for the typeclasses. In this new syntax `x → y` means that "x uniquely determines p". As shown in Figure 4, `Convert a b` typeclass is just a binary relation on types while the new `Add m n p` typeclass relates the type parameters such that m and n determine p. We can also have multiple functional dependencies as with TEq typeclass where t and u determine each other. In general we can have multiple parameters on both sides of the arrow $(x_1, \ldots, x_m) \rightarrow (y_1, \ldots, y_m)$.

```
class Convert a b where
   to :: a → b

class TEq t u | t → u, u → t

class Add m n p | m n → p where
  (+) :: m → n → p
instance Add Int Int Int where
   ...

instance Add Int Float Float where
   ...
instance Add Float Int Float where
   ...
instance Add Int Float Int -- Error!

e :: (Add Int Float b, Add b Int c) ⇒ c
e = (1 + 2.0) + 3
```

Fig. 4. Add with Functional Dependency

With these functional dependency annotations, the programmer can now accurately specify the intention of the typeclasses. Further, the compiler now has a way to detect inconsistencies with the declared instances and flag an error whenever it detects one. This improves error reporting as it flags errors at inconsistent declaration site rather previously confusing at the operator use site. For example, the functional dependency on `m n → p` can now help the compiler flag the instance `Add Int Float Int` as a conflicting instance `Add Int Float Float` because `Int` and `Float` together should uniquely determine a type.

There is also an additional advantage of using functional dependencies, they can be used to help the type inference improve the inferred types. The improvement is in the sense that seemingly polymorphic type could potentially resolve to a unique type as seen with the type of the expression `(2 + 3.0) + 3`. We can infer that type `b` has to be `Float`, and using this information we can then deduce that the type of the expression `c` has to be `Float`. Without the functional dependency on the class, it would not be impossible to make such an inference.

```
data Z
data S n

class IsPeano c
instance IsPeano Z
instance IsPeano n ⇒ IsPeano (S n)

class (IsPeano m, IsPeano m, IsPeano p)
    ⇒ Plus m n p | m n → p
instance IsPeano m ⇒ Plus Z m m
instance Plus n m p ⇒ Plus (S n) m (S p)
```

```
instance TEq Int Int
instance TEq Z Z
instance Plus (S Z) (S Z) m
    ⇒ TEq (S (S Z)) m
instance Plus (S Z) (S Z) m
    ⇒ TEq (S (S (S Z))) m -- Error!
```

Fig. 5. Peano Arithmetic with Functional Dependencies

With functional dependencies at our disposal, We can even perform peano arithmetic at type level as shown in Figure 5. First we define two datatypes `Z` and `S n` that represent the number zero and successor of a number `n`, respectively. The instances of `IsPeano` asserts that that `Z` is a peano number also, if `n` is a peano number then `S n` is a Peano number. The instances of `Plus` typeclass relates three peano numbers where the relation holds if the first two peano numbers add up to be equal to the third. Thus, `Peano Z m m` asserts `0 + n = n` while `Plus n m p ⇒ Plus (S n) m (S p)` asserts that (1 + n) + m = (1 + p) if n + m = p. The `TEq` instance holds when there is an appropriate `Plus` instance defined. Notice how the functional dependency `t → u, u → t` describes the fact that all instance of `TEq` should have the same type for `t` and `u`. Without this functional dependency it would impossible to enforce the criteria for a type class whose instances intended to mean that the types are equivalent.

The idea of functional dependency has been borrowed from the theory of databases[Armstrong 1974; Codd 1970].

3.3 Formalizing Type classes with Functional Dependencies

[Jones 1994] uses a general framework of qualified types to formalize the system with type classes. In this framework, the predicates on types are part of the syntax of the type language. The typeclasses are nothing but predicates on types and declaring instances is the assertion that the types satisfy the predicate. We say the $P \mid \Gamma \vdash M : \tau$ is a judgment that

holds when there is a typing derivation that shows M has type τ with predicates P being satisfied and the free variables in M are given types by the typing environment Γ. The goal of the type inference is to find the most general pair, (P, τ) for a given term M and typing environment Γ. The principle type for the term can then be given as $\forall\alpha.P \Rightarrow \tau$ where α are the type variables that appear in τ.

In set a simple theoretic semantics, type classes can be considered as a set of types. `Incable` = $\{$`Int`, `Char`$\}$ while `Group` = $\{$`Int`$\} \cup \{(\tau_1, \tau_2) \mid \tau_1, \tau_2 \in$ `Group`$\}$ and `Semigroup` = $\{$`Int`$\} \cup \{(\tau_1, \tau_2) \mid \tau_1, \tau_2 \in$ `Semigroup`$\}$. Multiparameter typeclasses represent relations over types. Thus, `TEq` = $\{$(`Int`, `Int`), (`Z`, `Z`)$\} \cup \{$(`S u`, `S t`) $\mid$ (`u`, `t`) $\in$ `TEq`$\}$

Term Variables	x, y		
Type Variables	α, β	Data Declarations	$T\overline{\alpha} = \overline{D\overline{\alpha}}$
Type constants	ι	Class Declarations	$\texttt{class}\ \overline{C\overline{\alpha}} \Rightarrow C\overline{\alpha} \mid \overline{\alpha} \rightarrow \alpha; \overline{x :: \tau}$
Class Constructors	C	Instance Declarations	$\texttt{instance}\ \overline{C\overline{\alpha}} \Rightarrow C\overline{\tau}; \overline{x = M}$
Data Constructors	D		
Type Constructors	T		

Types	τ	$::= \alpha \mid \iota \mid \tau \rightarrow \tau \mid \forall\alpha.\tau \mid \overline{C\alpha} \Rightarrow \tau \mid T\overline{\tau}$
Terms	M, N	$::= x \mid MN \mid \lambda x.M \mid \text{let } x = M \text{ in } N$

Fig. 6. Type classes with Functional Dependencies

3.4 Implementing Improvement and Consistency

We would want to have an algorithm that can detect inconsistent instance declarations using the the functional dependencies.

4 TYPE FAMILIES

Type family is a powerful mechanism of describing computation on types in a more natural style of equations. They are types that are parameterized by other types and specify when two types are equal using type level equations. For example, we can now define addition over the two previously mentioned types `Z` and `S n` as shown in Figure 8. This style is a lot more palpable for programmers who are already used to writing equations at term level. It also has a cleaner view and has less code clutter.

4.1 Open Type Families and Associated Types

The origins of type families can be traced back to another potential solution for relating types. In this scheme, each class would optionally get an additional associated type and the instance would instantiate it to a concrete type as shown in Figure 9. The `Add` type class has an associated `Result` type that depends on the type parameters of the class. This solution comes at a price of making the core language more complex. We would have to introduce a type equality

$$(\rightarrow \text{I})\ \frac{P \mid \Gamma \vdash M_1 : \tau_1 \quad P \mid \Gamma \vdash M_2 : \tau_2}{P \mid \Gamma \vdash (M_1, M_2) : \tau_1 * \tau_2}$$

Fig. 7. Typing judgments

```
type family Add n m
type instance  Add Z     m = m
type instance  Add (S n) m = S (Add n m)
```

Fig. 8. Peano Arithmetic with Type Family

constraints (`a ~ b`) into the type language and introduce a coercion term into the term language. The coercion term is a witness to the fact that the type equality that holds is valid.

```
class Add m n where
   type Result m n
   (+) :: m → n → Result m n

instance Add Int Int where
   type Result Int Int = Int
   (+) = intAdd
```

```
instance Add Int Float where
   type Result Int Float = Float
   i + f = addFloat (int2Float i) f

instance Add Int Float where
   type Result Int Float = Int -- Error!
   i + f = addInt i (float2int f)
```

Fig. 9. Associated Type Family

4.2 Closed Type Families

Consider an open type family `Eq u u` as shown in Figure 10. This is inconsistent as equations overlap. A solution for allowing such overlapping instances, is by declaring closed type families. The compromise we make is by enforcing that closed type families cannot be extended. The rewriting for such a type family is performed my matching equations from top to bottom.

```
data TT
data FF
```

```
type family TEq a b
type instance TEq a a = TT
type instance TEq a b = FF -- Error!
```

```
type family TEq a b where
    TEq a a = TT
    TEq a b = FF  -- Ok
```

Fig. 10. Open and Closed `TEq a b` Type family

4.2.1 Apartness of types. Overlapping can be understood in terms of apartness criterion. Apartness can be defined formally as follows:

Definition 1 (Apartness). Types τ_1 and τ_2 are apart if there exists no substitution S such that $S\tau_1 = S\tau_2$

4.3 Constraint Type families

In the previous account of open and closed type families, we have made an implicit assumption— the type families are all total, in the sense their domain is all the types. In the semantics of open type families, we leave the door open for the programmer to add type family instances while in closed type families we do not. What about the case where a closed

type family is not defined for particular instance? For example consider the closed type family shown in Figure 11. We know that `PTyFam Bool` has no satisfying equations associated with it and never will in the future. Hence, we should be able to reject such a type right away.

```
type family PTyFam a where          pty :: PTyFam Bool -- OK?
    PTyFam Int = Bool               pty = undefined
```

Fig. 11. Partial Closed Type Family

5 CONCLUSION AND FUTURE WORK

From a birds eye view, typeclasses with functional dependencies and type families are trying to achieving the same goal—computation on types. While functional dependencies have a flavor of relations, type families have a flavor of equations. This equational notation for type families seems to be one of the reasons why type families have gained more popularity in recent years. A formal proof about the equality of the expressive power of functional dependencies and type families is still an open problem[]

It is possible to encode type computations that are non-terminating in both type families and simple type classes.

```
type family Loop                    class Loopy [a] ⇒ Loopy a
type instance Loop = [Loop]
loop :: Loop                        loop :: Loop a ⇒ a
```

Fig. 12. Non-terminating Type Computation

As a feature, non-termination of a type checker may not be a very useful one. After all, we do want the type checker to terminate and say if a program is welltyped or not.

Another valid question is if functional dependencies can be emulated using type families, thus each functional dependency declaration can be syntactically transformed into a type family. This would significantly simplify the core of the language that might want to support both, functional dependencies and type families. There has been some recent work[Karachalias and Schrijvers 2017] in this direction, however, it remains an open problem and there has been no implementation of the formalization yet.

REFERENCES

William Ward Armstrong. 1974. Dependency Structures of Data Base Relationships.. In *IFIP congress*, Vol. 74. Geneva, Switzerland, 580–583.

E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

Mark P. Jones. 1994. *Qualified Types: Theory and Practice.* Cambridge University Press, Cambridge, UK.

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)*. Springer-Verlag, Berlin, Germany, 230–244.

Georgios Karachalias and Tom Schrijvers. 2017. Elaboration on functional dependencies: functional dependencies are dead, long live functional dependencies. *ACM SIGPLAN Notices* 52, 10 (2017), 133–147. https://doi.org/10.1145/3156695.3122966

TODO. 2050. Needs Citation. In *What do I know (Some series)*. Unknown, Somwhere, 0.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89)*. ACM, Austin, Texas, USA, 60–76. https://doi.org/10.1145/75277.75283