Imperial College London

Department of Computing

Higher-order Type-level Programming in Haskell

Supervisor
Prof. Susan Eisenbach

Csongor Kiss

Second Marker
Dr. Anthony Field

June 18, 2018

Abstract

Haskell, as implemented by the Glasgow Haskell Compiler (GHC), provides rich facilities for performing computation in the language of types, allowing programmers to ensure sophisticated static properties about their programs. The type language, however, has several limitations, restricting the range of expressible programs. Namely, the type language is first-order: no higher-order type functions are possible.

This work aims to lift this restriction, enabling higher-order type-level programming. We motivate the problem by demonstrating the power of the type language, and explain the limitation. Then we describe a solution, and present an implementation in GHC. We discuss the interaction with the current type system, and present novel ways of writing type-level programs.

Acknowledgements

I would like to thank my supervisor, Susan Eisenbach, and Tony Field for the motivating discussions and for keeping me on track. A particular thanks goes to Richard Eisenberg, who offered help and helpful insights about the topic.

Contents

1 Introduction **5**
1.1 Towards type safety . . . 6
1.2 Type-level properties . . . 6
1.3 Type families . . . 8
1.4 Limitations . . . 8
1.5 Contributions . . . 10

2 Background **11**
2.1 Types and kinds . . . 11
2.2 Rich kinds . . . 13
2.3 Equality . . . 14
2.4 Type functions are first-order . . . 17
2.5 Singletons . . . 19

3 Unsaturated type families **21**
3.1 A new arrow . . . 21
3.2 Type classes for type families . . . 23
3.3 Code reuse . . . 26
3.3.1 Subtyping . . . 26
3.3.2 Matchability polymorphism . . . 27
3.4 Type system . . . 28
3.4.1 Expressions . . . 28
3.4.2 Coercions . . . 29
3.4.3 Matchabilities . . . 30
3.4.4 Types and kinds . . . 30
3.4.5 Axioms . . . 30
3.5 Type classes . . . 30
3.5.1 Equality . . . 32

Roles . . . 35
3.5.2 Instance resolution . . . 37

4 Implementation **41**
4.1 Constraint-based type inference . . . 41
4.1.1 Constraints . . . 41
4.1.2 The solver pipeline . . . 42
Canonicalisation . . . 43
Flattening . . . 45
Canonicalising equalities . . . 45
Top-level interaction . . . 45
4.1.3 Unsaturated type families . . . 45

5 Unsaturated type families in practice **47**
5.1 Generic programming . . . 47
5.1.1 Algebraic structure . . . 48
5.1.2 Type-level traversals over the generic structure . . . 50
5.2 Type-level Generic programming . . . 51
5.3 Stateful programming . . . 54

6 Discussion **57**
6.1 Dependent Haskell . . . 57
6.2 Future work . . . 58

Chapter 1

Introduction

Types help us write programs by making sure that programming errors are detected long before we attempt to execute them. The more sophisticated the type system, the more bad programs it rejects, and the more good programs it allows. Haskell is a purely functional programming language, which means that functions are referentially transparent: they are not allowed to mutate values or perform side-effects. While referential transparency already eliminates a large class of erroneous programs (such as those resulting from careless mutation of global variables), there are still many ways in which programs can fail. For example, consider the *head* function, which returns the first element of a list:

```
  -- pre: non-empty list
head :: [a] → a
head (x : xs) = x
```

When all is well, *head* readily extracts the item in the first position[1]:

```
> ghci>  head [1 .. 10]
> 1
```

However, if we try to interrogate the empty list, we are faced with a *runtime error*:

```
> ghci>  head []
>  ⋆ runtime error : non − exhaustive patterns in function head ⋆
```

This is rather unfortunate, as the promise of static types was that they help us catch errors like these before we ever get to run our programs. However, not all is lost. We can write a safer version of *head*, which at least handles empty lists, and is explicit about when no value could be returned.

[1] Lines beginning with `ghci>` describe queries typed into the Glasgow Haskell Compiler's interactive REPL, and the following line is the corresponding result

1.1 Towards type safety

$$
\begin{aligned}
&\mathit{saferHead} :: [\,a\,] \to \mathit{Maybe}\ a \\
&\mathit{saferHead}\ (x : xs) = \mathit{Just}\ x \\
&\mathit{saferHead}\ [\,] = \mathit{Nothing}
\end{aligned}
$$

saferHead returns a value of type *Maybe a*. Users of the function must then pattern match on the result to discover whether the extraction succeeded (*Just x*) or not (*Nothing*).

```
> ghci>  saferHead []
> Nothing
```

While we have at least managed to avoid runtime exceptions, this solution is still unsatisfactory. Even if we *know* that the input list is non-empty, we still have to pattern match on the result every time we wish to use *saferHead*. Ideally the compiler would just tell us that we tried to call *head* on the empty list, and reject the program. In vanilla Haskell98 [Peyton Jones, 2003], *saferHead* is the best we can do. Over the long course of Haskell's history, however, the language has been extended with a plethora of type system features that allow programmers to explain their informal knowledge (such as that the input list has to be non-empty) to the typechecker by means of a more expressive vocabulary in the type language.

1.2 Type-level properties

Let us revisit the *head* function by using the full power of Haskell's type system, as implemented in the Glasgow Haskell Compiler (GHC)[2]. The requirement we wish to enforce is that the input list is non-empty, or in other words, that its length is at least one. For the built-in list type, the length of a list is not apparent in its type. For example, $[\,] :: [\,\mathit{Int}\,]$ and $[1, 2] :: [\,\mathit{Int}\,]$ both have the same type. In order to reason about the lengths, we need to reflect a notion of numeric values in the type system. To illustrate this consider the following inductive definition of natural numbers:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Nat} \\
&\quad = \mathit{Zero} \\
&\quad \mid \mathit{Succ}\ \mathit{Nat}
\end{aligned}
$$

[2] As of writing, the most recent released version is 8.4.3

At the value level, this introduces the *data* constructors *Zero* and *Succ*. The `DataKinds` language extension [Yorgey et al., 2012] allows the *Zero* and *Succ* data constructors also to be used at the type level as *type* constructors. We can then create a version of list, which we call *Vector*, that is indexed by the number of elements it contains.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Vector} :: \mathit{Nat} \to \star \to \star\ \textbf{where} \\
\quad \mathit{VNil}\ \ :: \qquad\qquad\qquad\qquad \mathit{Vector}\ \ '\mathit{Zero}\ a \\
\quad \mathit{VCons} :: a \to \mathit{Vector}\ n\ a \to \mathit{Vector}\ (\,'\mathit{Succ}\ n)\ a
\end{array}
$$

Here, *Vector* is a *generalised algebraic data type (GADT)* [Cheney and Hinze, 2003], which means that its constructors refine the type index. The empty constructor *VNil* simply represents a list whose length is *'Zero*[3]. *VCons* takes a value of type a and another vector of length n, and constructs a vector of length one bigger than n. Now we can specify the the *safeHead* function as

$$
\begin{array}{l}
\mathit{safeHead} :: \mathit{Vector}\ (\,'\mathit{Succ}\ n)\ a \to a \\
\mathit{safeHead}\ (\mathit{VCons}\ x\ xs) = x
\end{array}
$$

That is, it takes a vector whose length is one bigger than some natural number (i.e at least one). Since the only way to produce an index headed by *'Succ* is by using *VCons*, calling *safeHead* with the empty list is a type error:

```
> ghci> safeHead (VCons 1 (VCons 2 VNil))
> 1
> ghci> safeHead VNil
> ⋆type error : Couldn't match type 'Zero with ' Succ n0 ∗
```

Now it's impossible to try to extract the head of the empty list, as the compiler will stop us from doing so.

1.3 Type families

GADTs are a powerful tool, but working with them can be awkward. For example, it's not obvious how to carry out certain operations on them that modify their lengths, such as appending two vectors together, as in order for this to be expressible, we need a way of performing computation (addition in this case) at the type level. With the `TypeFamilies`

[3] The ' (tick) symbol is used as a syntactic aid to signify that *Zero* is used in a type context here.

extension [Chakravarty et al., 2005], we can express type functions (called type families) in a natural way.

```
type family Add (a :: Nat) (b :: Nat) :: Nat where
  Add 'Zero b     = b
  Add ('Succ a) b = 'Succ (Add a b)
```

The *Add* type family carries out the addition of two *Nat*s by doing recursion on the first. This allows us to express the type of *append*, that is: it takes two vectors and returns a new vector whose length is the sum of the two inputs.

```
append :: Vector n a → Vector m a → Vector (Add n m) a
append VNil v           = v
append (VCons a as) v = VCons a (append as v)
```

1.4 Limitations

Type families, or type functions, enable a natural and convenient way of expressing type-level computations. However, the current implementation of type families is not ideal, in that there are some computations that can be expressed at the value level that cannot be mirrored at the type level. In particular, functions at the value level (i.e. regular Haskell functions) and functions at the type level (type families) have syntactic differences [Eisenberg and Stolarek, 2014]. Most importantly, value-level functions can be curried (or partially applied), while type families can not; this is due to assumptions made during type inference.

To illustrate this, recall the definition of the *map* function in Haskell:

```
map :: (a → b) → [a] → [b]
map _ []       = []
map f (x : xs) = f x : map f xs
```

map is *higher-order*: it takes a function as its argument, which it then applies to every element of a list to produce a new list of the results.

```
> ghci> map (1+) [1, 2, 3, 4]
> [2, 3, 4, 5]
```

Here, (1+) is a partially applied function: we have provided a single argument to $(+) :: Int \to Int$, resulting in a new function of type $Int \to Int$ that takes only one argument. We could try and define the type-level equivalent of *map* as follows:

```
type family Map (f :: a → b) (xs :: [a]) :: [b] where
  Map _ '[] = '[]
  Map f (x ': xs) = f x ': Map f xs
```

Map operates on the promoted list type (hence the use of the ' symbol in the constructors). While this definition of *Map* is perfectly sensible, we cannot pass in any type functions as arguments, because such type functions can only be constructed via the partial application of a type family. Unfortunately the current implementation of type families in Haskell does not allow such partial application type families are required to be saturated meaning that the only way to call them is by providing every argument at once. As an example, the he following expression is rejected:

```
> ghci> :kind Map (Add ('Succ 'Zero)) '['Zero]
```

where *Add* is the type family defined above. The problem here is that *Add* is given a single argument, but its arity is 2. This means that in *Add* ('*Succ* '*Zero*), *Add* is partially applied, or unsaturated, and as such, not allowed by the type system. What this means is that the type language is first-order. To map a function over a list, we have to specialise *Map* to every possible use-case:

```
type family MapAddOne (xs :: [Nat]) :: [Nat] where
  MapAddOne '[]         = '[]
  MapAddOne (x ': xs) = 'Succ x ': MapAddOne xs
```

This results in code duplication, and in practice reluctance to use type-level programming techniques as the process of specialising higher-order functions to first-order ones is too laborious.

In this project, we show how this restriction can be lifted, enabling higher-order programming at the type-level. This work brings the expressive power of the type language closer to the term language, and provides an important step towards bringing full-spectrum dependent types to Haskell. [Weirich et al., 2017]

1.5 Contributions

The main contributions of this work are as follows:

- We provide a detailed description of the problem, and an evaluation of the limitations of Haskell's existing type system.
- We develop an extension to Haskell's type system called `UnsaturatedTypeFamilies` that enables partial application of type-level functions in a backwards-compatible manner. In particular, we show that type inference does not suffer as a result of enabling the feature.
- We present a technique for type classes to be instantiated with type families, and describe a conservative equational theory for instance resolution.
- We develop a formalisation of the new type system and present a proof of consistency.
- We evaluate the new features, and detail a case study that shows how this new style of type-level programming can be applied to reduce substantially the volume of redundant code in a widely-used Haskell library.
- We detail an implementation of `UnsaturatedTypeFamilies` in GHC, and discuss the interaction with the existing constraint solver.

We conclude by discussing the road to Dependent Haskell, and how `UnsaturatedTypeFamilies` fits into that road.

Chapter 2

Background

The Haskell programming languages has a long history, dating back to the late 1980s, when a group of researchers decided to put an end to the proliferation of lazy purely functional research programming languages, and unify them into one that can then be uniformly studied [Hudak et al., 1992]. Since its inception, the language has been extended along several dimensions, from syntactic conveniences through new runtime representations to novel type system features. To give some context for this work, we begin with a tour of several language extensions that are implemented in the Glasgow Haskell Compiler, an industrial-strength optimising compiler for Haskell. We direct our focus on features that enrich the type system with more reasoning power, giving a more in-depth account of ones mentioned in Chapter 1.

2.1 Types and kinds

Static type systems ensure some degree of correctness by imposing restrictions on the kind of programs that the compiler should accept. For example, it doesn't make sense to apply the $\mathit{even} :: \mathit{Int} \to \mathit{Bool}$ function to an argument whose type is *String*, as a *String* value can't be used where an *Int* value is expected. This line of reasoning can be extended to the types themselves: some combinations of type do not make sense. For example, the type *Int Int* (*Int* applied to itself) is clearly bogus, as the *Int* type takes no arguments. An example of a type that does take an argument is *Maybe*:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

Maybe is a *type constructor*: it takes a type argument and returns a new type. The *Maybe Int* type is the result of applying *Maybe* to *Int*. There are other type constructors that take two, three, etc. arguments.

Analogously to the way *types* classify *values*, *types* are classified by *kinds*. All value types (such as *Int*, *String*, etc.) have kind $\star$ (pronounced *star*, or simply *type*). Thus the kind of *Maybe* is $\star \to \star$. The *Maybe* type constructor abstracts over types, but there are ways of abstracting over type constructors too. As an example, consider the *Functor* type class from the Haskell standard library:

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

Types that are instances of *Functor* can be mapped over: their inner a can be replaced with a b by a suitable function. The *Maybe* data type represents the possibility of a non-existent value, and it readily admits a *Functor* instance, mapping over the a in the *Just* constructor:

```
instance Functor Maybe where
  fmap :: (a → b) → Maybe a → Maybe b
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

The reason we could implement a sensible *Functor* instance for *Maybe* is that it is parameterised by another type a, which can be mapped over. *Functor* is *higher-kinded*: it abstracts over type constructors. We can provide a *kind signature* to *Functor* to make the expected kind more explicit (although this is inferred from the use of $f\ a$).

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

The base kind system in Haskell 98 [Peyton Jones, 2003] consists only of two formation rules:

$$\boxed{\kappa ::= \star \mid \kappa \to \kappa}$$

That is, everything is either a value type of kind $\star$, or an arrow between two kinds. An important property of $\star$ is that it is *open*: new inhabitants can be added at any time simply by creating new data types:

```
data Wurble
```

We get a new type *Wurble* that also inhabits $\star$.

2.2 Rich kinds

The base kind system distinguishes between value types and type constructors, but it is rather weak when we wish to encode more complicated patterns. Kiselyov et al. [2004] describe a mechanism to store information about collections in their types, such as the length of the list. To achieve this, they represent natural numbers in the following way:

```
data HZero
data HSucc n
```

They specify that *HZero* means zero, and *HSucc n* is the successor of n. For instance, the number two is represented as *HSucc* (*HSucc HZero*). Neither types have any constructors, which means that it's not possible to construct a value of type *HZero*. The sole purpose of the type is to exist at the type level. This formulation is rather weakly-kinded, however, as the kind of *HSucc* is $\star \to \star$, which means that as far as the type checker is concerned, *HSucc Bool* is a perfectly valid type, even though it makes no sense in our intuition of natural numbers. Yorgey et al. [2012] enriched the kind system by enabling programmers to define their own kinds with corresponding type constructors. The mechanism is called *promotion*, as it re-purposes ("promotes") ordinary data constructors to be used in types. Recall the *Nat* type from 1.2:

```
data Nat
  = Zero
  | Succ Nat
```

Nat extends the set of kinds, and has two inhabitants, $'Zero :: Nat$ and $'Succ :: Nat \to Nat$. In this new scheme, the type $'Succ\ Bool$ is ill-kinded, as *Bool* has kind $\star$ instead of *Nat*. The essence of promotion is enriching the syntax of kinds by lifting syntactic elements from the term language into the kind language.

Types classify *values*, and *kinds* classify *types*. It is natural to as the question: how are *kinds* classified? What's the "type" of *Nat*, or the "type" of $\star$? Martin-Löf [1984] introduces the notion of *universes* to classify types. Universes are further collected into universes. This leads to an infinite universe hierarchy each indexed by an ordinal: $\mathcal{U}_0 \in \mathcal{U}_1 \in \mathcal{U}_3 \in ...$

As a simplification, Weirich et al. [2013] introduced the $\star :: \star$ (pronounced "type in type") axiom into the type system. This axiom collapses the infinite universes into one, which means that $Nat :: \star$, and $\star :: \star :: \star :: ....$ Thus, $\star$ is the set of all sets, which leads to the inconsistency of the type system (as it can express Russell's paradox). This means Haskell

cannot be used as a logic system, but this was already the case due to non-termination of type reductions, as we will see.

2.3 Equality

Up to this point, the only way for two types to be "equal" was either either syntactic equivalence, or being related by a type synonym:

```
type String = [Char]
```

We assigned a new name, *String*, to the type [*Char*]. Type synonyms are a useful aid in readability, as the new alias can convey some special meaning for a given type.

However, type synonyms always describe a trivial rewrite rule. Indeed, one of the first things the type checker does is expanding type synonyms to make reasoning about type equality simpler. Since these always describe a single rule, no conditional expansion is possible, and recursive type synonyms are disallowed.

Chakravarty et al. [2005] extended the type system with non-trivial type equalities. Such type equalities are introduced into the equational theory as axioms by type family declarations. For example, we can define a type family that relates containers with the type of elements they contain. Then, we can implement generic functions that operate over collections and their elements:

```
type family Elem c
class Collection c where
  empty :: c
  insert :: Elem c → c → c
```

Here, *Collection* is a type class that ranges over some collection c. The described operations are *empty*, which returns an empty collection, and *insert*, which takes a single element and a collection, and returns a new collection with the element inserted. Note that the type of elements in c is represented by *Elem* c. Two possible instances are:

```
type instance Elem [a] = a
instance Collection [a] where
  empty      = []
  insert x xs = x : xs
```

```
type instance Elem Text = Char
instance Collection Text where
  empty      = emptyText
  insert x xs = textCons x xs
```

Where $[a]$ is a list that contains value of type a, and *Text* is some opaque efficient string representation that contains *Char*s.

The equations **type instance** *Elem* $[a] = a$ and **type instance** *Elem Text* = *Char* introduce *axioms* into the equational theory. They declare that *Elem* $[a]$ equals a and *Elem Text* equals *Char* respectively. Equality of types in Haskell is written with the $\sim$ symbol: *Elem* $[a] \sim a$. For example, if we ask GHCi what the type of *insert* `'c'` (that is, inserting the character 'c') is, it will show us the following:

```
> ghci> :type insert 'c'
> insert 'c' :: (Collection c, Elem c ~ Char) => c -> c
```

That is, *insert* `'c'` works for any collection whose elements are *Char*s. If we wish to specify that we're interested in the type of inserting values to *String* collections (which is a synonym for $[Char]$), we can do so by providing an explicit type argument as input:

```
> ghci> :type insert @String
> insert @String :: Char -> String -> String
```

Eisenberg et al. [2016] introduced *visible type applications*, a mechanism for explicitly specifying which instantiation of a type parameter we wish to use for a polymorphic function. This way, we can provide types as *inputs* to functions. The order in which type parameters have to be provided follows their topological ordering in the type signature (the first to appear is the first argument, etc.). In this case, there was only one argument, c.

Because they allow to perform non-trivial reductions, type families are also known as type functions. A type family is indexed by some type arguments, and it encodes a family of rewrite rules that depend on these arguments. Eisenberg et al. [2014b] extended type families with the ability to define a closed set of overlapping equations. To illustrate this, we define the following type family:

```
type family Equals (a :: k) (b :: k) :: Bool where
  Equals a a = 'True
  Equals a b = 'False
```

Equals demonstrates several properties of type families. Firstly, they can take polymorphically kinded arguments (here, parameterised by some kind variable k). Secondly, non-linear patterns: the first equation mentions the same variable twice, and it matches only when the two arguments are equal, in which case it returns (the promoted) *'True*. Finally, that equations can overlap: the third equation technically matches any combination of types

(as long as they are of the same kind), but since this type family is *closed* (all equations are provided at the same time, indicated by the **where** after the kind signature), equations can overlap and are tried in a top-to-bottom order.

Type families can also be recursive. recall the *Add* type family from Section 1.3:

```
type family Add (a :: Nat) (b :: Nat) :: Nat where
  Add 'Zero b = b
  Add ('Succ a) b = 'Succ (Add a b)
```

Add is parameterised by two types of kind *Nat*. If the first type argument is *'Zero*, then we return the second (0 + b = b). Otherwise, we recurse in the first argument, eventually terminating as the first argument reaches zero. This definition of *Add* introduces two top-level axioms:

```
Add 'Zero b     ~ b
Add ('Succ a) b ~ 'Succ (Add a b)
```

We can prompt GHCi to evaluate type-level additions for us (in this case 1 + 1):

```
> ghci> :kind (Add ('Succ (Zero)) ('Succ (Zero)))
>  = 'Succ ('Succ 'Zero)
```

In this case, recursion is *structural*, and we can easily see that *Add* will terminate for any inputs. However, we note that it is possible to encode non-terminating types:

```
type family Bottom where
  Bottom = Bottom
```

Thus, the type language truly is a capable system for doing arbitrary computation. Encoding programs in the language of types is referred to as *type-level programming*. After giving a taste of the power of the type system, we now describe one of its major limitations.

2.4 Type functions are first-order

As alluded to in the introduction, type families cannot take other type families as arguments, rendering the type language a first-order system. We now make precise the reason for this restriction, and describe existing solutions.

The source of the problem is the type inference engine. When faced with an equality between type applications, the type checker decomposes them into its constituent parts. To illustrate this, consider the (somewhat contrived) function

$$\begin{aligned}&\mathit{replaceWith10} :: \mathit{Functor}\ f \Rightarrow f\ a \to f\ \mathit{Int}\\&\mathit{replaceWith10}\ \mathit{fa} = \mathit{fmap}\ (\backslash\_ \to 10)\ \mathit{fa}\end{aligned}$$

replaceWith10 replaces all the values in some container f with 10, using the *Functor* instance for the container. Suppose we called it as such: *replaceWith10* (*Just False*). How does the type checker know what to instantiate the type variables of *replaceWith10* with? We know that the input has to be of the shape $f\ a$, and we're given *Maybe Bool*. That is, we have to solve the equality constraint $f\ a \sim \mathit{Maybe\ Bool}$. The type checker proceeds by *decomposing* this equality into two smaller equalities, $f \sim \mathit{Maybe}$ and $a \sim \mathit{Bool}$. These can now be turned into a substitution map $[f \mapsto Maybe, a \mapsto \mathit{Bool}]$, and type inference is done: the type of *replaceWith10* here is $\mathit{Maybe\ Bool} \to \mathit{Maybe\ Int}$. The way we could proceed was by taking apart the application $f\ a$.

Now consider an example with hand-written equalities:

$$\begin{aligned}&\mathit{decompose} :: (f\ a \sim g\ b) \Rightarrow a \to b\\&\mathit{decompose} = \mathit{id}\end{aligned}$$

The type of *decompose* is $a \to b$, but it is implemented with *id*, whose type requires both the input and output to be the same. *decompose* type checks, because we provided a given equality $f\ a \sim g\ b$, which the typechecker decomposes to learn that $f \sim g$ and $a \sim b$. The first derived equality is discarded as no f and g are manifest in the rest of the type signature. $a \sim b$, however, is used to refine the result type into $a \to a$, which is now of course compatible with the implementation *id*.

We show that allowing type families to be unsaturated in this system violates type safety. First, we define a type family which discards its argument and always returns the type *Int*.

$$\begin{aligned}&\textbf{type}\ \mathit{family}\ \mathit{AlwaysInt}\ a\ \textbf{where}\\&\quad \mathit{AlwaysInt}\ \_ = \mathit{Int}\end{aligned}$$

Now we instantiate the type variables in *decompose* explicitly with visible type applications. The order in which the type variables appear is f, a, g, b, so this is the order in which we supply them:

$$bad :: Integer \to Char$$
$$bad = decompose\ @AlwaysInt\ @Integer\ @AlwaysInt\ @Char$$

Type type of *bad* is $(AlwaysInt\ Integer \sim AlwaysInt\ Char) \Rightarrow Integer \to Char$. The equality constraint normalises to $Int \sim Int$ (as *AlwaysInt* reduces), which trivially holds, so the function can be called.

Thus, we could convince the type system that the *id* function can coerce any *Integer* into any *Char*. Of course, this violates type safety, as running the code could lead to segmentation faults at runtime, because the size of *Integer* is different from that of *Char*.

In order to avoid this, GHC rejects the *bad* program. The root cause of the problem is that we try to substitute a *non-injective* function in for a variable that GHC assumes to be injective (and *generative*).

Definition (Injectivity)**.** f *is* injective $\iff f\ a \sim f\ b \implies a \sim b$.

Definition (Generativity)**.** f *is* generative $\iff f\ a \sim g\ b \implies f \sim g$.

Eisenberg [2016] calls type functions that are both injective and generative *matchable*:

Definition (Matchability)**.** f *is* matchable $\iff$ *it is* injective *and* generative.

Type families, in general, are neither *injective*, nor *generative*. The only way the type system can safely assume that $f\ a \sim g\ b$ can be decomposed into $f \sim g$ and $a \sim b$ is by ensuring that neither f nor g can be instantiated with type families. This is achieved by enforcing that type families are always fully saturated, so they can reduce. This technical reason explains why type families can not appear without all of their arguments applied to them: that would wreak havoc with type inference, as decomposing unreduced families would give rise to bogus equality constraints.

2.5 Singletons

Eisenberg and Weirich [2012]' singletons library sidesteps this problem by differentiating type functions and type constructors at the *kind level*. However, they achieve this by *defunctionalisation* using code generation generation [Sheard and Peyton Jones, 2002]. Defunctionalisation eliminates higher-order functions and replaces them with a first-order *apply* function. All higher-order function calls are replaced by a unique identifier which represents that function cal. Then, the *Apply* function dispatches on the unique identifier.

For example, unsaturated applications of the constant *Const* function and the identity *Id* can be represented as data types:

```
type TyFun a b = a → b → ⋆
infixr 0 `TyFun`
data SymConst2          :: ⋆ `TyFun` ⋆ `TyFun` ⋆
data SymConst (n :: ⋆) :: ⋆ `TyFun` ⋆
data SymId              :: ⋆ `TyFun` ⋆
```

Here, we had to define all possible unsaturated applications of the constant function: in the first case, where it is applied to no arguments (so it takes two), and in the second case, applied to a single argument. *Apply* dispatches on the defunctionalisation symbols:

```
type family Apply (f :: a `TyFun` b) (x :: a) :: b where
  Apply SymConst2 n     = SymConst n
  Apply (SymConst n) _ = n
  Apply SymId n         = n
```

Currying is explicitly written out in the *SymConst2* case. Then we can write a version of *Map* that takes a defunctionalisation symbol, and applies it to every element of a list with *Apply*.

```
type family MapDefun (f :: a `TyFun` b) (x :: [a]) :: [b] where
  MapDefun f '[] = '[]
  MapDefun f (x ': xs) = Apply f x ': MapDefun f xs

> ghci> :kind MapDefun (SymConst Bool) '[Int, String, Char, String]
>  = '[Bool, Bool, Bool, Bool]
```

Executing this approach is extremely tedious. Defining a new function means having to specify all of its possible partial applications (which is the same number as the number of arguments the function takes), and modifying the *Apply* function to handle it appropriately. Indeed, the singletons library provides a code generator tool that generates these definitions for us. Still, this results in namespace pollution (as now our environment will contain all of the defunctionalisation symbols), and overall bad ergonomics.

Therefore we seek a first-class solution: introducing something like *TyFun* and *Apply* into the type system, but without doing defunctionalisation. The important property of *TyFun* is that is syntactically a different arrow from →, which is the kind of type constructors.

Chapter 3

Unsaturated type families

We show how to extend Haskell's type system to support unsaturated type families. First we present the programmer's view of the feature, then we formalise our type system.

3.1 A new arrow

The most important user-facing change is that the kinds of type constructors is distinguished from the kinds of type functions. We borrow the notation proposed by Eisenberg and Weirich [2012] in the singletons library. That is, the arrow of type constructors (matchable) is $\to$, and the arrow of type functions (unmatchable) is $\twoheadrightarrow$.

Correspondingly, we introduce a new application form, which applies an argument to an unmatchable function, written f `@@` x. The use of (`@@`) is not necessary in the surface syntax when f's kind is known, as it can be inferred.

Recall the *Map* type family from Section 1.4, but now with an unmatchable function as its argument:

```
type family Map (f :: a ↠ b) (xs :: [a]) :: [b] where
  Map _ '[] = '[]
  Map f (x ': xs) = f x ': Map f xs
```

Map can now be called with a partially applied type family, as its argument's kind suggests[1]:

```
> ghci> :kind Map (Add ('Succ 'Zero)) '[('Succ 'Zero), 'Zero]
>  = '[('Succ ('Succ 'Zero)), 'Succ 'Zero]
```

To demonstrate a practical use-case for this feature, we define a data type that represents a person (as a product of their name and age), parameterised by some type function.

[1] All code examples in this report are type checked with my fork of GHC, with the help of the Lhs2TeX tool

```
data Person f = MkPerson
  { name :: f @@ String
  , age  :: f @@ Int
  }
```

We can think of the *Person* type as a schema that can be instantiated to different shapes based on the supplied function f. The simplest example is providing the *Id* type family, which essentially erases the f:

```
person :: Person Id
person = MkPerson "Toby" 21
```

The real utility of this parameterisation of course comes from plugging in more interesting functions. For example, suppose that we want to serialise the *Person* type for writing to a database, which only supports certain primitive types. We could establish a mapping between the types in the schema and the primitives with a type family:

```
type family DbPrimitive a
type instance DbPrimitive String = DbText
type instance DbPrimitive Int    = DbInt
```

We could then write a function with the signature

```
serialise :: Person Id → Person DbPrimitive
serialise (MkPerson name age) = MkPerson (mkDbText name) (mkDbInt age)
```

Or perhaps a pretty-printable version of *Person*, that contains only *String* values:

```
pretty :: Person Id → Person (Const String)
pretty (MkPerson name age) = MkPerson (show name) (show age)
```

Where *Const* is the constant function:

```
type family Const a b where
  Const a b = a
```

General higher-order programming techniques are also applicable, of course. For example, we can now implement a type-level sorting algorithm in a very concise way by reusing *Filter*, a function which drops all elements of a list that don't match a given predicate:

```
type family Filter (f :: k ↠ Bool) (xs :: [k]) :: [k] where
  Filter _ '[] = '[]
  Filter f (x ': xs) = If (f x) (x ': Filter f xs) (Filter f xs)
type family QuickSort xs where
  QuickSort '[] = '[]
  QuickSort (x ': xs) =
    QuickSort (Filter (≤ x) xs) ++ '[x] ++ QuickSort (Filter (> x) xs)
```

3.2 Type classes for type families

Haskell supports ad-hoc overloading of functions via its type class mechanism [Wadler and Blott, 1989]. Recall the *Functor* type class that classifies types that can be mapped over.

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

The type f is called the *head* of the constraint (in general, for type classes with multiple parameters, the set of parameters is referred to as the head). *Functor* allows us to write overloaded functions that operate on container-like things without having to specify which container we mean, as long as its elements can be mapped over:

```
addOne :: Functor f ⇒ f Int → f Int
addOne xs = fmap (+1) xs
```

The *Functor* class can then be instantiated to various container-like types:

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just x

instance Functor [] where
  fmap _ []       = []
  fmap f (x : xs) = f x : fmap f xs
```

Crucially, *Functor* classifies type constructors that take one type argument. Suppose we wanted to use the *addOne* function, but only with a single *Int*, and not some container of *Int*s. As the signature of *addOne* requires that the input is of the form f *Int*, we can not just pass in a bare *Int* value. The traditional solution to this problem is introducing a trivial container and instantiating it to *Functor*:

```
newtype Identity a = Identity a
unwrap :: Identity a → a
unwrap (Identity a) = a
instance Functor Identity where
  fmap f x = Identity (f (unwrap x))
```

The *Functor* instance for *Identity* does nothing but unwraps the container, applies the function, then wraps it back again. We can now call *addOne* by wrapping our *Int* into *Identity*, running *addOne*, then unwrapping the result:

```
addOneInt x = unwrap (addOne (Identity x))
```

Doing so each time we wish to use a function that is parameterised over some *Functor* is tedious. In fact, this pattern is so ubiquitous that Maguire [2018] created a dedicated compiler plugin to automatically resolve conversions between *Identity* a and a, allowing to use *addOne* directly on *Int*s. However, their solution is unsound: it introduces an axiom that *Identity* $a \sim a$, but by clever construction of equality proofs, their inequality can also be observed. It also only handles the special case of *Identity*. We seek a more general (and sound) solution to this problem.

If we could instantiate *Functor*'s f to the *Id* type family, then we would not need to do any wrapping or unwrapping:

```
instance Functor Id where
  fmap :: (a → b) → a → b
  fmap f x = f x
```

The unwrapping happens implicitly as *Id* a and *Id* b reduce to a and b respectively. However, *Id* appears unsaturated here, which means that the existing type class mechanism rejects the above code. Now that we have lifted the saturation restriction, we encounter a different problem: the program doesn't kind-check. *Functor* requires a type constructor (of kind $\star \to \star$), but *Id* has kind $\star \twoheadrightarrow \star$.

However, with the kind-level distinction of type families and type constructors, it is valid at least syntactically to define a version of the *Functor* class that ranges over non-generative type abstractions.

```
class Functor′ (f :: ⋆ ↠ ⋆) where
  fmap′ :: (a → b) → f @@ a → f @@ b
```

Suppose we wish to instantiate the class with the identity type family Id. Substituting in Id for f and reducing the equations, we arrive at the following definition:

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor'}\ \mathit{Id}\ \textbf{where}\\ \quad \mathit{fmap'} :: (a \to b) \to a \to b\\ \quad \mathit{fmap'}\ f\ x = f\ x\end{array}$$

But how would a call to $\mathit{fmap'}$ be resolved? Instance resolution is the process of solving a constraint involving a type class by instantiating the constraint's head with a given type and matching it against every instance of the class. When a matching instance is found, the corresponding implementation is selected. For example, when calling $(\mathit{show} :: \mathit{Show}\ a \Rightarrow a \to \mathit{String})$ on a value of type Int, we instantiate the constraint to $\mathit{Show}\ \mathit{Int}$ and find an existing instance in scope. Notice, however, that the type of $\mathit{fmap'}$ is ambiguous in the $\mathit{Functor'}\ \mathit{Id}$ instance: Id itself is not manifest. To demonstrate the problem, consider the following use of $\mathit{fmap'}$:

$$\mathit{fmap'}\ (+1)\ 10 :: \mathit{Int}$$

There is no way to determine which instance of $\mathit{Functor'}$ was used to produce the result. From $\mathit{fmap'}$'s signature, we can deduce that the argument 10's type must be of the form $f\ a$, for some f and a. Furthermore, from knowing that $(+1)$ has type $\mathit{Int} \to \mathit{Int}$ we can deduce that a must be Int. Thus we're left with the problem of solving $f\ \texttt{@@}\ \mathit{Int} \sim \mathit{Int}$. That is, given that some function f applied to Int results in Int, what is f?

To address instance resolution at the use sites, we recall the problematic application from before: $\mathit{fmap'}\ (+1)\ 10 :: \mathit{Int}$. The only way to proceed in a predictable manner is by requiring the user to declare which instance she meant. This is done with a visible type application: $\mathit{fmap'}\ \texttt{@}\mathit{Id}\ (+1)\ 10 :: \mathit{Int}$. Here, the constraint solver starts off with a given equality of $f \sim \mathit{Id}$ as it tries to solve $f\ \texttt{@@}\ \mathit{Int} \sim \mathit{Int}$, which can now happily proceed by substituting Id for f and discharging the remaining constraint by reflexivity. This means that f is not inferred, but this doesn't break existing type inference, as before it was not possible to have an equality constraint with an unsaturated type function on either side.

3.3 Code reuse

There is something unsatisfactory about the $\mathit{Functor'}$ class: modulo matchability, it is defined in the same way as the original $\mathit{Functor}$. We discuss two alternatives for eliminating this code duplication.

3.3.1 Subtyping

Eisenberg [2016], proposes a subtype relationship between $\to$ and $\twoheadrightarrow$, with $\to$ being the subtype of $\twoheadrightarrow$. This would allow the matchable $\to$ arrow to be used in contexts where an unmatchable $\twoheadrightarrow$ arrow is required. While conceptually simple, this solution results in weak type inference. To illustrate this, consider the following function call:

$$\mathit{fmap}'\ (+1)\ x :: \mathit{Maybe}\ \mathit{Int}$$

What's the type of x? From the signature of fmap' we can conclude that it's of shape $f\ \texttt{@@}\ a$. Furthermore, $(+1)$ has type $\mathit{Int} \to \mathit{Int}$, therefore a is Int. Similarly to before, we're faced with the problem of $f\ \texttt{@@}\ \mathit{Int} \sim \mathit{Maybe}\ \mathit{Int}$, with the unknown f. Since f's kind is $\star \twoheadrightarrow \star$, we cannot decompose the equality over the applications, so we're stuck. While this does not make existing code fail to type check, it certainly reduces the ergonomics of the type class: users now have to choose between type inference (*Functor*) and code reuse (*Functor′*).

3.3.2 Matchability polymorphism

As we have seen, subtyping does not play well with type inference. We turn to parametric polymorphism: instead of having two arrows, we have a single arrow, called `ARROW` that is parameterised over its matchability.

$$\begin{array}{l}\textbf{data}\ \mathit{Matchability} = \mathit{Matchable} \mid \mathit{Unmatchable}\\ \textbf{type}\ \texttt{ARROW} :: \mathit{Matchability} \to \star \to \star \to \star\end{array}$$

Then the two arrows $\to$ and $\twoheadrightarrow$ simply become synonyms for the two possible instantiations of `ARROW`:

$$\begin{array}{l}\textbf{type}\ (\to) = \texttt{ARROW}\ \ '\mathit{Matchable}\\ \textbf{type}\ (\twoheadrightarrow) = \texttt{ARROW}\ \ '\mathit{Unmatchable}\\ \textbf{type}\ (\texttt{@@}) :: \mathit{Arrow}\ m\ k\ j \to \mathit{Arrow}\ m\ k\ j\end{array}$$

This allows us to define *FunctorP* class, which is the matchability-polymorphic version of *Functor*:

$$\begin{array}{l}\textbf{class}\ \mathit{FunctorP}\ (f :: \texttt{ARROW}\ m\ \star\ \star)\ \textbf{where}\\ \quad \mathit{fmapP} :: (a \to b) \to f\ a \to f\ b\end{array}$$

Then we can instantiate *FunctorP* with both type constructors and type families:

instance *FunctorP Maybe* **where** ...
instance *FunctorP Id* **where** ...

During type inference, the typechecker generates metavariables to be filled in by the constraint solver. In order to get predictable type inference, we make the following rule:

Matchability defaulting: at the end of constraint solving, all unsolved matchability metavariables are defaulted to *matchable*.

Let us revisit the example from before to see how this works in practice. We are trying to infer the type of x:

$$\mathit{fmapP}\ (+1)\ x :: \mathit{Maybe\ Int}$$

From the signature of *fmapP* we gather that x's type is of shape $f\ \texttt{@@}\ a$, and from $(+1) :: \mathit{Int} \to \mathit{Int}$ that $a \sim \mathit{Int}$. Thus the type inference problem is $f\ \texttt{@@}\ a \sim \mathit{Maybe\ Int}$. As before, we're stuck at this point. However, the kind of f is $\mathit{Arrow}\ m \star \star$ with some metavariable m. Since we can not proceed further with constraint solving, the **matchability defaulting** rule applies, and we conclude that $f :: \star \to \star$. Our new constraint is thus $f\ \mathit{Int} \sim \mathit{Maybe\ Int}$, which can now be decomposed over the applications to get the wanted constraint $f \sim \mathit{Maybe}$, which can be solved with unification, arriving the the solution $x :: \mathit{Maybe\ Int}$.

This means that *fmapP* behaves exactly like *fmap* (the original) does, when no visible type application is used. However, we have the ability to provide a type argument, and use an unmatchable instance:

$$\mathit{fmapP}\ \texttt{@}\mathit{Id}\ (+1)\ x :: \mathit{Int}$$

Here, the constraint is once again $f\ \texttt{@@}\ \mathit{Int} \sim \mathit{Int}$, but with a given constraint that $f \sim \mathit{Id}$. This results in the matchability of f's arrow to be unified with *unmatchable*, thus allowing the type family to reduce. Thus, we have inferred that x's type must be *Int*.

3.4 Type system

Haskell is a big language with several type system features that make it difficult to reason about the correctness of compilation. To ensure that the high-level features are translated sensibly, GHC uses an explicitly typed core calculus as its internal representation, called GHC Core [Eisenberg, 2015]. Core is an implementation of Sulzmann et al. [2007]'s System

Metavariables:

x term	a, b type	c coercion
C axiom	F type family	m matchability

e	$::=$	$\lambda x : \tau.e \mid \Lambda a : \kappa.e \mid \lambda c : \phi.e \mid e\ \gamma \mid e_1\ e_2 \mid e\ \tau \mid e \triangleright \gamma$	terms
τ, σ	$::=$	$a \mid \tau_1 \texttt{@@}^{\mu} \tau_2 \mid \forall a : \kappa.\tau \mid H$	types
κ	$::=$	$\star \mid \kappa_1 \to^{\mu} \kappa_2$	kinds
H	$::=$	$(\to) \mid (\Rightarrow) \mid (\sim^{\kappa}) \mid F$	type constants
ϕ	$::=$	$\tau \sim^{\kappa} \sigma$	propositions
Φ	$::=$	$[\overline{a : \kappa}].\tau \sim \sigma$	axioms
γ, η	$::=$		coercions
		$\mid$ **refl** $\tau \mid$ **sym** $\gamma \mid$ **trans** $\gamma_1\ \gamma_2$	equivalence
		$\mid \gamma_1 \texttt{@@}^{\mu} \gamma_2 \mid \forall a : \kappa.\gamma$	congruence
		$\mid c \mid C(\overline{\tau})$	assumption
		$\mid$ **left** $\gamma \mid$ **right** $\gamma \mid \gamma \,@\, \tau$	decomposition
μ, ν	$::=$	$m \mid M \mid U$	matchability
Γ	$::=$	$\varnothing \mid \Gamma, a : \kappa \mid \Gamma, c : \phi \mid \Gamma, x : \tau \mid \Gamma, m : \mu$	typing contexts

FIGURE 3.1: Syntax

F_C, which itself is an extension of System F (the polymorphic lambda calculus) with explicit type-equality witnesses (coercions). Here, we present a subset of F_C, in particular, we omit data types, as they are orthogonal to our feature. Figure 3.1 displays the syntax of this subset of F_C, extended with matchability information about type functions. Now we explain the components of the type system.

3.4.1 Expressions

The $\lambda x : \tau.e$ and $\Lambda a : \kappa.e$ terms are the traditional term and type abstraction forms of the polymorphic lambda calculus, with respective term application $e_1\ e_2$ and type application $e\ \tau$. The innovation of System F_C is the use of *coercions*, or type equalities (made more precise in Section 3.4.2). A coercion $\phi = \tau \sim^{\kappa} \sigma$ represents an equality between τ and σ at kind κ. Coercions can be abstracted over with $\lambda c : \phi.e$ and applied as $e\ \gamma$. For example, the function

$$co :: (a \sim Int) \Rightarrow a \to Int$$
$$co = id$$

abstracts over a proof that $a \sim Int$. In order to call it, the caller must supply the appropriate witness by building the coercion. In practice, GHC's type inference algorithm produces these witnesses, so coercion holes are filled in automatically.

Finally, $e \triangleright \gamma$ represents a *cast*: a conversion between two types whose equality is witnessed by some coercion γ.

3.4.2 Coercions

Non-syntactic equalities (such as those introduced by type family equations) pose a great challenge in compilation, as they make it difficult to do sanity checks on the intermediate representation. Coercions solve this problem by reifying the typing derivations and encoding them into the terms themselves. What this means is that the only way to convert an expression e of type τ_1 into type τ_2 is by providing a witness (a coercion) γ of the type equality $\tau_1 \sim \tau_2$ and explicitly *casting* e by γ: $e \triangleright \gamma$. The corresponding typing rule is E_Cast in Figure 3.4. Figure 3.5 displays the formation rules for coercions. They are a syntactic reification of the equivalence relationship (with corresponding reflexivity (Co_Refl), symmetry (Co_Sym), and transitivity (Co_Trans) rules) with congruence. This way, type checking in F_C is syntactic, as all the derivations are encoded in the terms via casts. For example, in order to use a coercion $\gamma : Bool \sim a$ to prove that $e : a$ has type $e : Bool$, we cast e as such: $e \triangleright \textbf{sym}\ \gamma$. For a more comprehensive account of the type system, refer to Sulzmann et al. [2007]. In particular, we omit the operational semantics here, which is the expected reduction rules for the lambda calculus, with "push" rules for the coercions, to ensure they don't get in the way of evaluation.

3.4.3 Matchabilities

We enrich the syntax with matchability information. M stands for matchable, U stands for unmatchable, and m represents a matchability meta-variable (for matchability polymorphism).

3.4.4 Types and kinds

We modify the original System F_C by parameterising the arrow kind and the type application form with their matchability. Correspondingly, the coercion decomposition rules Co_Left and Co_Right are only allowed to decompose matchable applications. Notably, the Ty_App kinding rule allows the application form to be reused for both kind of matchabilities. Rule Ty_Fun assigns an unmatchable arrow kind to type functions.

$$\boxed{\vdash \Gamma}$$

$$\frac{}{\vdash \varnothing}\textsc{Empty} \qquad \frac{\vdash \Gamma \qquad a\#\Gamma}{\vdash \Gamma, a : \kappa}\textsc{TyVar} \qquad \frac{\Gamma \vdash \tau \sim \sigma : \star \qquad c\#\Gamma}{\vdash \Gamma, c : \phi}\textsc{CoVar}$$

$$\frac{\Gamma \vdash \tau : \star \qquad x\#\Gamma}{\vdash \Gamma, x : \tau}\textsc{Var} \qquad \frac{\Gamma \qquad m\#\Gamma}{\vdash \Gamma, m : \mu}\textsc{MVar}$$

FIGURE 3.2: Context validity

$$\boxed{\Gamma \vdash \tau : \kappa}$$

$$\frac{\vdash \Gamma \qquad a : \kappa \in \Gamma}{\Gamma \vdash a : \kappa}\textsc{Ty\_Var} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_1 \to^{\mu} \kappa_2 \qquad \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \tau_1 \texttt{@@}^{\mu} \tau_2 : \kappa_2}\textsc{Ty\_App}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash (\to) : \star \to^{M} \star \to^{M} \star}\textsc{Ty\_Arrow} \qquad \frac{\vdash \Gamma}{\Gamma \vdash (\Rightarrow) : \star \to^{M} \star \to^{M} \star}\textsc{Ty\_CoArrow}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash (\sim^{\kappa}) : \kappa \to^{M} \kappa \to^{M} \star}\textsc{Ty\_Equality} \qquad \frac{\vdash \Gamma \qquad F : [\overline{a : \kappa'}].\kappa}{\Gamma \vdash F : \overline{\kappa'} \to^{U} \kappa}\textsc{Ty\_Fun}$$

FIGURE 3.3: Type kinding

3.4.5 Axioms

Axioms (Φ) are top-level equalities. Valid contexts only have type family applications on the left-hand side of these equalities, and axioms have no free variables (axioms that have a non-empty set of binders are called axiom schemes).

3.5 Type classes

We omit type classes from our type system, as they are largely orthogonal to type families, but we discuss them here, because they raise interesting questions about the equational theory. Type class instances must be coherent, which means that for any given type, there must be at most one matching instance. Therefore it's not sufficient to find a matching

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\vdash \Gamma \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\text{E\_VAR} \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x : \tau.e : \tau \to \sigma}\text{E\_ABS}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma}\text{E\_APP} \qquad \frac{\Gamma, a : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda a : \kappa.e : \forall a : \kappa.\tau}\text{E\_TABS}$$

$$\frac{\Gamma \vdash e : \forall a : \kappa.\sigma \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e\ \tau : \sigma[\tau/a]}\text{E\_TAPP} \qquad \frac{\Gamma, c : \sigma_1 \sim \sigma_2 \vdash e : \tau}{\Gamma \vdash \lambda c : \sigma_1 \sim \sigma_2.e : \phi \Rightarrow \tau}\text{E\_CABS}$$

$$\frac{\Gamma \vdash e : (\sigma_1 \sim \sigma_2) \Rightarrow \tau \qquad \Gamma \vdash \gamma : \sigma_1 \sim \sigma_2}{\Gamma \vdash e\ \gamma : \tau}\text{E\_CAPP}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash e \triangleright \gamma : \tau_2}\text{E\_CAST}$$

FIGURE 3.4: Expression typing

instance, but we also have to make sure that no other instances match. Doing so amounts to proving that all the other instance heads are *apart* from the candidate: that is, they are not equal. For this, we need a definition of what equality means.

3.5.1 Equality

Suppose we defined the type family Id_2 in the following way:

```
type family Id2 a where
  Id2 a = a
```

Notice that the definition for Id_2 looks identical to that of Id. The central question is this: is Id the "same" as Id_2? Suppose then we wrote a corresponding $Functor'$ instance:

```
instance Functor' Id2 where
  fmap' :: (a → b) → a → b
  fmap' f x = f x
```

Should this be allowed? If not, on what basis should it be rejected? If allowed, is there a way to infer either instance from the use of $fmap'\ (+1)\ 10 :: Int$? Either way, we can

$$\boxed{\Gamma \vdash \gamma : \phi}$$

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \textbf{refl}\ \tau : \tau \sim \tau}\textsc{Co\_Refl} \qquad \frac{\Gamma \vdash \gamma : \sigma \sim \tau}{\Gamma \vdash \textbf{sym}\ \gamma : \tau \sim \sigma}\textsc{Co\_Sym}$$

$$\frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \tau_2 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim \tau_3}{\Gamma \vdash \textbf{trans}\ \gamma_1\ \gamma_2 : \tau_1 \sim \tau_3}\textsc{Co\_Trans} \qquad \frac{\vdash \Gamma \qquad c : \tau \sim \sigma \in \Gamma}{\Gamma \vdash c : \tau \sim \sigma}\textsc{Co\_Var}$$

$$\frac{\begin{array}{c}\Gamma \vdash \gamma_1 : \tau_1 \sim \sigma_1 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim \sigma_2 \\ \Gamma \vdash \gamma_1 \mathbin{@@^{\mu}} \gamma_2 : \kappa \qquad \Gamma \vdash \sigma_1 \mathbin{@@^{\mu}} \sigma_2 : \kappa\end{array}}{\Gamma \vdash \gamma_1 \mathbin{@@^{\mu}} \gamma_2 \sim \sigma_1 \mathbin{@@^{\mu}} \sigma_2}\textsc{Co\_App}$$

$$\frac{\begin{array}{c}\Gamma \vdash \gamma : \tau_1 \mathbin{@@^{M}} \tau_2 \sim \sigma_1 \mathbin{@@^{M}} \sigma_2 \\ \Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \sigma_1 : \kappa\end{array}}{\Gamma \vdash \textbf{left}\ \gamma : \tau_1 \sim \sigma_1}\textsc{Co\_Left} \qquad \frac{\begin{array}{c}\Gamma \vdash \gamma : \tau_1 \mathbin{@@^{M}} \tau_2 \sim \sigma_1 \mathbin{@@^{M}} \sigma_2 \\ \Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \sigma_1 : \kappa\end{array}}{\Gamma \vdash \textbf{right}\ \gamma : \tau_2 \sim \sigma_2}\textsc{Co\_Right}$$

$$\frac{\begin{array}{c}C : [\overline{a : \kappa}].\sigma_1 \sim \sigma_2 \\ \vdash \Gamma \qquad \Gamma \vdash \tau : \kappa\end{array}}{\Gamma \vdash C(\overline{\tau}) : \sigma_1[\overline{\tau/a}] \sim \sigma_2[\overline{\tau/a}]}\textsc{Co\_Axiom}$$

$$\frac{\Gamma \vdash \gamma : \forall a : \kappa.\tau_1 \sim \forall a : \kappa.\sigma_1 \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash \gamma \mathbin{@} \tau : \tau_1[\tau/a] \sim \sigma_1[\tau/a]}\textsc{Co\_Inst}$$

Figure 3.5: Formation rules for coercions

not answer these questions until we can make a decision on whether or not Id and Id_2 are equal. We seek an equational theory that meets the following criteria:

Decidable apartness: Instance resolution requires that there is exactly one candidate instance, and concluding this requires a decidable algorithm for the apartness judgement between any two types. Many dependent type theories use $\beta\delta\eta$-convertibility as their equality relation. That is, checking that two types normalise to the same term (up to α-equivalence). The normalisation steps are the usual β and η reductions from pure lambda calculus, extended with δ-reduction, which corresponds to expansion of constants, such as type family equations in our case. We can show that Id and Id_2 are $\beta\delta\eta$-convertible:

$$\begin{aligned} Id &\equiv_\eta \lambda x.Id\ x && \eta\text{-expansion} \\ &\equiv_\delta \lambda x.x && \delta\text{-reduction of } Id\ x \end{aligned}$$

$$\begin{aligned} Id_2 &\equiv_\eta \lambda y.Id_2\ y && \eta\text{-expansion} \\ &\equiv_\delta \lambda y.y && \delta\text{-reduction of } Id_2\ y \end{aligned}$$

$$\lambda x.x \equiv_\alpha \lambda y.y$$

However, defining equality like this is only decidable if $\beta\delta\eta$-reduction is strongly normalising. That is, if all reduction sequences terminate with a normal form (i.e. a term that can no longer be reduced). Most dependently typed languages employ a totality checker to ensure that all terms have a normal form. For example, Brady [2013]'s *Idris* language refuses to evaluate terms at compile-time that are not known to be total in order to keep type checking decidable.

Haskell has no such totality checker[2], and indeed with relative ease one can define a non-terminating type family:

```
type family Loop a where
  Loop a = [Loop a]
```

We conclude that $\beta\delta\eta$-convertibility is undecidable in this context.

[2]Technically it does, but it is so restrictive that even simplistic (and obviously terminating) type-level programs are rejected, thus programmers tend to disable it.

Forwards compatibility: Weirich et al. [2017]'s specification for System D (the implicitly typed core calculus for Dependent Haskell) includes a non-trivial definitional equality that contains the reduction relation (β). As of writing, it has no η rules, but there is a real possibility of adding them. This would mean that Id_2 and Id are equated in their scheme. As discussed above, this definitional equality is undecidable (but the type system remains sound as equality propositions are formulated outside of the syntax of types). Their equality is a conservative extension of today's Haskell, as only new types it equates do not exist today (as type-level lambdas are not part of the language today). Our notion of type equality ought to be compatible with that of Dependent Haskell, to ensure that no program will fail to typecheck in the future.

The alert reader will notice that these two requirements, *decidability* and *compatibility*, are at odds with each other. On the one hand, a decidable theory is desired for type class instance selection, but the future plans for Haskell mandate an undecidable definitional equality for type functions. To resolve this conflict, we introduce a separate notion of equality that applies only when selecting instances. Crucially, this equality is different from, and is allowed to coexist with the definitional equality of Dependent Haskell.

Roles

Having different notions of equality in different contexts is not unprecedented. Haskell already has various different ways of equating types. Indeed, there are currently three different type equalities in GHC today, classified by the *role*-system [Breitner et al., 2016]. Thus, when declaring that two types are the same, we mean they are equal at one of the following roles:

Nominal equality This is the traditional equality that we normally reason about in the type system. Two types a and b are nominally equal, written $a \sim_N b$, either if they have the same name (such as $Bool \sim_N Bool$), or there is a nominal equality axiom relating them. Such axioms are introduced by type family equations. For example, consider the type family

$$\begin{array}{l}\textbf{type}\ \mathit{family}\ \mathit{AlwaysBool}\ a\ \textbf{where}\\ \quad \mathit{AlwaysBool}\ \_ = \mathit{Bool}\end{array}$$

AlwaysBool introduces an axiom scheme that for all a, $\mathit{AlwaysBool}\ a \sim_N \mathit{Bool}$. These axioms are used in δ-reduction when equating two types. In today's Haskell,

nominal equality is the same as δ-convertibility. Instance selection happens with nominal equality: for all classes C, if $a \sim_N b$ and there is an instance $C\ a$, then there must be an instance $C\ b$, and the two instances must be the same.

Representational equality Two types a and b are representationally equal ($a \sim_R b$), when they share the same bit-pattern at runtime, i.e. their runtime representations are identical. Representational equalities are generated by **newtype** constructors:

newtype *Age* = *Age Int*

Age is "free" in the sense that it does not introduce an extra layer of indirection, in fact, it is completely erased during compilation. The definition for *Age* gives rise to the representational equality $Age \sim_R Int$. However, note that $Age \not\sim_N Int$ (this distinction is the reason for using **newtype**s, after all). Representational equality can be used to coerce values between two types without incurring any operational cost. For example, suppose we wanted to turn a list of *Int* values into a list of *Age* values. We could write *map Age xs*, which would traverse the list and apply the *Age* constructor to every element. However, as *Age* is erased, this operation is a no-op, and a very costly one at that (still requires a linear-time traversal of the list). Instead, we can use the *coerce* function, which lifts the $Int \sim_R Age$ equality to $[Int] \sim_R [Age]$ by congruence over type applications.

Phantom equality Any two types are equal at the phantom role ($Int \sim_P Bool$). Phantom equality extends representational equality for types with unused type parameters:

newtype *Phantom a* = *Phantom Int*

Since the a parameter is not used on the right-hand side, its inferred to have a **phantom** role in representational equalities involving the *Phantom* type. For example, $Phantom\ Bool \sim_R Phantom\ String$, despite $Bool \not\sim_R String$ (but $Bool \sim_P String$).

Published formalisation of Dependent Haskell do not discuss the role system, but it is assumed that their equality is an extension of the **nominal** one, which is the one used for instance selection today. We propose a new equality role **class** to be used for instance selection.

Definition (Class role)**.** *Two types a and b are equal at role **class**, written $a \sim_C b$, if and only if they are $\beta\delta$-convertible. That is, they can be made syntactically equal by reducing type families in them.*

$$\boxed{\tau \rightsquigarrow \sigma}$$

$$\frac{}{\tau \rightsquigarrow \tau}\textsc{Red\_Refl} \qquad \frac{\tau_1 \rightsquigarrow \sigma_1 \qquad \tau_2 \rightsquigarrow \sigma_2}{\tau_1 \mathbin{@@^{\mu}} \tau_2 \rightsquigarrow \sigma_1 \mathbin{@@^{\mu}} \sigma_2}\textsc{Red\_App}$$

$$\frac{\tau \rightsquigarrow \sigma}{\forall a : \kappa.\tau \rightsquigarrow \forall a : \kappa.\sigma}\textsc{Red\_Forall} \qquad \frac{C : [\overline{a : \kappa}].\tau_1 \sim \tau_2}{\tau_1[\overline{\sigma/a}] \rightsquigarrow \tau_2[\overline{\sigma/a}]}\textsc{Red\_Axiom}$$

FIGURE 3.6: Type reduction

Since we have no type lambdas as of yet (so β-reduction is not possible), **class** equality coincides with the **nominal** equality. We omit roles from our presentation to simplify the metatheory. All equalities are assumed to be at a nominal role (δ). In the next section, we impose certain restrictions on instance heads, and prove that under these restrictions, instance resolution is decidable.

3.5.2 Instance resolution

Figure 3.6 displays an auxiliary non-deterministic type reduction relation in the ambient context. The only way for the relation to make a non-trivial step (i.e. one that isn't just reflexivity) is if there is a matching axiom in the context which corresponds to a type family equation, via the rule RED_AXIOM. For example, *Maybe* (*Id Int*) steps to *Maybe Int* via RED_APP and RED_AXIOM. In the latter, we find that there is an axiom $Id\ a \sim a$, and that *Id Int* can be obtained from $(Id\ a)[a \mapsto Int]$. We say that type is in normal form if $\rightsquigarrow$-reduction to any other type is impossible (i.e. it contains no redexes).

In order for instance head resolution to be decidable, we make the following restriction on instance heads:

Definition (Instance head validity)**.** *A type τ is a valid instance head if:*

1. *It is quantifier-free.*
2. *It is in normal form.*
3. *All unmatchable applications are to known type families.*

In other words, the instance head is in normal form, and we wish to preserve normal forms after substitution. In general, application does not preserve normal forms. Even

if both f and a are in normal form, f `@@` a may still reduce if f is an unsaturated type family that takes a single argument (thus after the application it becomes saturated). In general, the terms that can be substituted for f in a way that preserves normal forms are called *neutral* terms. A naive approach would be to allow any f that takes more than one argument based on its kind. That is, $f :: k \twoheadrightarrow j \twoheadrightarrow l$ for some kinds k, j, and l.

However, this is not sufficient. Consider the following the class instances:

class *Bad* $(a :: \star \twoheadrightarrow \star)$
instance *Bad* (f `@@` *Int*)
instance *Bad* (f `@@` *Char*)

In both instances, we are matching on $f :: \star \twoheadrightarrow \star \twoheadrightarrow \star$ applied to some type. There is no guarantee that a type function of kind $\star \twoheadrightarrow \star \twoheadrightarrow \star$ takes two arguments to be fully saturated. For example, both F and G take only a single argument (and return a new function that takes one argument):

type *family* F $(a :: \star) :: \star \twoheadrightarrow \star$**where**
  F *Int* $=$ G *Char*
type *family* G $(a :: \star) :: \star \twoheadrightarrow \star$**where**
  G *Char* $=$ F *Int*

In the first instance of *Bad*, instantiating f to F would mean an instance lookup for *Bar* (F `@@` *Int*). This either matches the first instance, or reduces to *Bar* (G `@@` *Char*) and matches the second (which can then reduce again...). Therefore we must only allow known type families so that we can check that they are unsaturated to avoid stepping by any matching equations.

Theorem (Substitution)**.** *If* $\tau_1 \rightsquigarrow \tau_2$*, then* $\tau_1[\sigma/a] \rightsquigarrow \tau_2[\sigma/a]$

Proof. By induction. □

Theorem (Local confluence)**.** *If* $\tau \rightsquigarrow \sigma_1$ *and* $\tau \rightsquigarrow \sigma_2$ *then there is some* σ_3 *such that* $\sigma_1 \rightsquigarrow \sigma_3$ *and* $\sigma_2 \rightsquigarrow \sigma_3$

Proof. By induction on the structure of τ.

Case $\tau = a$**:** As axioms are only introduced by type family equations, they only match type family applications (thus no type variable can occur on the left-hand side of an

axiom). Therefore the only matching rule is Red_Refl. Thus $\sigma_1 = \sigma_2 = \tau$, and we can set $\sigma_3 = \tau$, and we're done. The only way this can step is by Red_Refl (as variables do not appear on the left hand side of axioms).

Case $\tau = \tau_1 \,@@^{\mu}\, \tau_2$: Case split on μ:

Case M: Axioms only have unmatchable applications on the left-hand side, so the only non-trivial step is by Red_App. By inversion on Red_App we get that $\sigma_1 = \sigma_{11} \,@@^{M}\, \sigma_{12}$ and $\sigma_2 = \sigma_{21} \,@@^{M}\, \sigma_{22}$ and $\tau_1 \rightsquigarrow \sigma_{11}$, $\tau_1 \rightsquigarrow \sigma_{21}$, $\tau_2 \rightsquigarrow \sigma_{12}$, and $\tau_2 \rightsquigarrow \sigma_{22}$. Then we are done by the induction hypothesis.

Case U: Unmatchable applications do appear on the left-hand side of axioms, but only if the type family is fully saturated. If τ is an under-saturated type family application, then the only way for it to step is via the Red_App rule. The proof then follows as in the previous case. We prove the saturated case by induction on the two reductions.

Case Red_App/Red_App: As above.

Case Red_App/Red_Axiom: We know that $\tau_1 \rightsquigarrow \sigma_{11}$ and $\tau_2 \rightsquigarrow \sigma_{12}$ by inversion on Red_App. The other case reduced with an axiom, so $\tau = F(\overline{\tau_0})$ for some type family F. Let the matching axiom be $C : [\overline{a : \kappa}]F(\overline{a}) \sim \sigma'$. Then $\sigma_2 = \sigma'[\overline{\tau_0/a}]$. Since $\tau = F(\overline{\tau_0})$, σ_{11} must be headed by F, and $\sigma_{11}\ \sigma_{12}$ is a fully saturated application of F, so they have to reduce with the same axiom C as σ_2 did. Then we appeal to the substitution lemma, and we're done.

Case Red_Axiom/Red_Axiom: Both reductions have to be by the same axiom, thus the resulting types have to be the same.

Case $\tau = \forall a : \kappa.\tau$: We are done by the induction hypothesis (and the substitution lemma).

□

Theorem (Instance head selection is decidable)**.** *Given a type τ such that $\tau \rightsquigarrow^* \sigma$, and σ is in normal form, and a class C a whose instance heads are all valid, instance selection for τ is decidable.*

Proof. Due to the assumption that all instance heads are valid, any substitution preserves normal forms, thus instance matching can be done syntactically. □

Chapter 4

Implementation

We describe the implementation of `UnsaturatedTypeFamilies` in GHC. We begin with a description of the type inference algorithm employed by GHC, and accordingly the required modifications to make unsaturated type families work.

4.1 Constraint-based type inference

Type inference is the process of elaborating Haskell code into GHC Core. GHC achieves this by means of a *constraint-based* type inference algorithm called OutsideIn(X) [Vytiniotis et al., 2011]. It is parameterised over the underlying constraint domain, X. This allows the type system to be extensible with domain-specific constraint solvers, such as one that solves units of measure equations [Gundry, 2015]. Here, we instantiate X to (potentially unsaturated) type families, and show how type checking happens in GHC.

GHC first generates constraints from the Haskell source code, then solves these constraints with the OutsideIn(X) algorithm. During constraint solving, GHC generates coercions which reify the typing derivation of each term in the program. These coercions are placed into the generated Core in the form of explicit type casts that justify the type conversion of two terms (see Section 3.4.2).

This section gives a high-level, somewhat simplified overview of the workings of OutsideIn(X), with a primary focus on the parts that are relevant in this project.

4.1.1 Constraints

Figure 4.1 shows the grammar of our constraint system. The only type of constraint is the equality constraint (and omit type class constraints from this presentation). Top-level axioms are those introduced by type family equations. The ξ meta-variable is used for types that do not mention type families (neither fully saturated, nor under-saturated).

Metavariables: ξ type function-free types

Q	$::=$	$\epsilon \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2$	constraints
$\mathcal{Q}$	$::=$	$Q \mid \mathcal{Q}_1 \wedge \mathcal{Q}_2 \mid \forall \overline{a}.F\ \overline{\xi} \sim \tau$	top-level axiom schemes
l	$::=$	$g \mid w$	constraint flavours

FIGURE 4.1: Syntax for constraint system

Furthermore, each constraint is either *given* (with a g flavour), or *wanted* (w). Terms are typed relative to some constraint: $Q; \Gamma \vdash e : \tau$. The typing judgements means that given the constraint Q and a type environment Γ, e has type τ. For example, the following function is checked with the given constraint $a \sim \mathit{String}, b \sim \mathit{Int}$:

$$\mathit{lengthy} :: (a \sim \mathit{String}, b \sim \mathit{Int}) \Rightarrow a \to b$$
$$\mathit{lengthy} = \mathit{length}$$

the corresponding judgement is

$$(a \sim \mathit{String}, b \sim \mathit{Int}); (\mathit{length} : \mathit{String} \to \mathit{Int}) \vdash \mathit{length} : a \to b$$

The first step of type inference is *constraint generation* which is a judgement of the form:

$$\Gamma \mapsto e : \tau \rightsquigarrow Q_w$$

That is, we infer that e has type τ in the type environment Γ, and we generate constraint Q_w.

4.1.2 The solver pipeline

After constraint generation, the next step is *constraint solving*, with the judgement

$$\mathcal{Q}\ ;\ Q_g \overset{simp}{\mapsto} Q_w \rightsquigarrow Q_r\ ;\ \theta$$

That is, it takes the top level set of axiom schemes ($\mathcal{Q}$), the set of given constraints (Q_g), and the set of wanted constraints (Q_w). It then produces a type substitution map θ, and a set of residual constraints (Q_r): these are constraints that could not be simplified further. If Q_r is ϵ (the trivial constraint), then the Q_w constraints are solved. Otherwise, the residual constraints are added to the given set, and the solver iterates.

Consider the following example:

$$\begin{aligned}&\mathit{weirdId} :: \mathit{Id}\ a \sim b \Rightarrow a \to b\\&\mathit{weirdId}\ x = x\end{aligned}$$

Here, we are given (from the context) that $\mathit{Id}\ a \sim b$, and the generated wanted is $a \sim b$ (from the function's definition).

$$\begin{aligned}&[g]\ \mathit{Id}\ a \sim b\\&[w]\ a \quad\ \sim b\end{aligned}$$

First we process the givens, and find that $\mathit{Id}\ a$ can be reduced to a, and our given constraint becomes $a \sim b$, which discharges the wanted, and we're done.

Essentially, the goal is to break apart the constraints into left-to-right rewrite rules that can then be repeatedly applied until we have found a solution. The complexity of the constraint solver arises from the fact that if done naively, this process could easily fail to terminate. For example, given an equality $a \sim \mathit{Id}\ a$, using it as a rewrite rule will result in continuous expansion of as into $\mathit{Id}\ a$s without making any progress. The constraint solver is carefully engineered to avoid situations like this. Here we review the most important steps, focusing on the treatment of type families.

The constraint solver maintains a work list, which is a priority queue of constraints to be processed. *Given* constraints are processed first, as they potentially generate useful information that can help solve the *wanted* items. In addition, GHC keeps track of an *inert* set, which stores all the constraints have no pairwise interaction. We say that two constraints interact with each other if they can be combined to deduce new information. For example, $a \sim \mathit{Bool}$ and $a \sim b$ interact, as we can learn that $b \sim \mathit{Bool}$. Maintaining the inert set is an important optimisation.

In order to avoid loops, constraints are turned into strict left-to-right rules, by a process known as *canonicalisation*.

Canonicalisation

The $\vdash_{\text{can}} Q$ judgement is defined by the following rules:

$$\boxed{\vdash_{\text{can}} Q}$$

$$\frac{tv \prec \xi \qquad tv \notin ftv(\xi)}{\vdash_{\text{can}} tv \sim \xi}\text{CEQ} \qquad\qquad \frac{}{\vdash_{\text{can}} F\ \overline{\xi} \sim \xi}\text{CFEQ}$$

According to rule CEQ, for an equality constraint involving a type variable on the left to be canonical, it must be the case that the right hand side is type-function free (ξ). The following partial order is defined on types:

F $\overline{\tau}$	$\prec$	τ	when $\tau \neq$ G $\overline{\tau}$
tv_1	$\prec$	tv_2	when $tv_1 \leqslant tv_2$ lexicographically
tv	$\prec$	ξ	

TABLE 4.1: Auxiliary ordering relation on types

This means that canonical constraints are oriented in a way that eliminates type family applications (as type families only appear on the left-hand side of canonical equalities). Note that rule $CFEQ$ requires that all type families be applied to type family-free types. This is achieved by *flattening*. A subset of the canonicalisation rules are seen in 4.2.

REFL	$canon[l]\ (\tau \sim \tau)$	$=$	$\{\epsilon, \epsilon\}$
APPDEC	$canon[l]\ (\tau_1\ \texttt{@@}^M\ \tau_2 \sim \tau_3\ \texttt{@@}^M\ \tau_4)$	$=$	$\{\epsilon, \tau_1 \sim \tau_2 \wedge \tau_3 \sim \tau_4\}\}$
OCCCHECK	$canon[l]\ (tv \sim \xi)$		
	where $tv \in \xi, \xi \neq tv$	$=$	$\bot$
ORIENT	$canon[l]\ (\tau_1 \sim \tau_2)$		
	where $\tau_1 \prec \tau_2$	$=$	$\{\epsilon, \tau_2 \sim \tau_1\}$
FFLATWL	$canon[w]\ (\mathbb{F}[G\overline{\xi}] \sim \tau)$	$=$	$\{\epsilon, (\mathbb{F}[\beta] \sim \tau) \wedge (G\overline{\xi} \sim \beta)\}$
FFLATGL	$canon[g]\ (\mathbb{F}[G\overline{\xi}] \sim \tau)$	$=$	$\{[\beta \mapsto G\overline{\xi}], (\mathbb{F}[\beta] \sim \tau) \wedge (G\overline{\xi} \sim \beta)\}$

TABLE 4.2: Canonicalisation rules

The canonicaliser $canon[l]$ produces a flattening substitution (see below), and a set of new constraints, whose flavour is l.

$\mathbb{F}$ represents a type with a hole in it (so $\mathbb{F}[\alpha]$ means that somewhere in the type there is a type α).

Flattening

All τ-types (which can contain type families) are replaced with ξ-types (type family-free types) by replacing each type family application $F(\overline{\tau})$ in τ with a fresh variable β, called a flattening meta-variable. In case of wanted constraints, a new constraint $F(\overline{\tau}) \sim \beta$ is generated which it then places on the work list. This will ensure that further occurrences of $F(\overline{\tau})$ are replaced with β. In case of given constraints, a new flattening substitution is generated $[\beta \mapsto G\overline{\xi}]$. This ensures that after solving $G\overline{\xi}$, all occurrences of β will be replaced with the result.

Canonicalising equalities

This is the stage where equalities are decomposed, by rule $APPDEC$. We ensure that only matchable applications can be decomposed in this way.

Top-level interaction

Finally, the work item is interacted with all the top-level axioms, allowing type family equations to reduce.

4.1.3 Unsaturated type families

We note that the $canon[l]$ function can not decompose unmatchable applications, but according the $\vdash_{\text{can}} Q$ judgement, no canonical constraints involve unmatchable application forms to unknown type functions (as $CFEQ$ requires the function to be known, and the actual implementation makes use of this assumption). We could extend the judgement with a new form of canonical constraint, $\vdash_{\text{can}} \xi_1 \texttt{@@}^U \xi_2 \sim \xi_3$. This would then have to be treated similarly to normal type family applications. Flattening would replace them with flattening metavariables, and generate flattening substitutions.

Instead, we can reuse the existing type family machinery. The only requirement for the $CFEQ$ rule is that the type family is known.

We define *Apply* as a special type family, whose first argument is an unknown unsaturated type family. Applications are of the form *Apply* $f\ \overline{\xi}$. When the type family becomes

known to be some F, the special form is $Apply\ F\ \overline{\xi}$ is rewritten to $F\ \overline{\xi}$. The only change we need to make is an extra flattening rule which replaces unmatchable applications to type variables with the *Apply* family, and another canonicalisation rule that replaces applications of *Apply* to known type families with the normal type family application form, so that the top-level interactions may happen.

Chapter 5

Unsaturated type families in practice

In the previous chapters, we have described the theory and implementation of the `UnsaturatedTypeFamilies` GHC extension. We now evaluate the extension and demonstrate its applicability in practice through a case study of the generic-lens library [Kiss et al., 2018], which uses advanced type-level programming techniques in its implementation. We show that with `UnsaturatedTypeFamilies`, the library's internals can be made significantly shorter, and more obviously correct. We begin with a short introduction into datatype-generic programming. The reasons for this are twofold: firstly, to demonstrate the library itself, and secondly, as a primer of a new style of type-level programming that we will use to improve the library's implementation.

5.1 Generic programming

"Boilerplate" code is code that describes something obvious or repetitive in a verbose way. It is usually a consequence of inadequate tooling that makes it impossible to explain the programmer's intent at a high-level, and instead requires spelling out the uninteresting details of the algorithm. For example, consider a function that visits every number in a data-structure and increments it by one. The high-level description of the function is rather straightforward, but implementing it takes some amount of effort, as we have to define how to actually locate the numbers in a given data type. In the following example, we define a data type with two fields of type *Int*:

$$\textbf{data}\ Pair = MkPair\ Int\ Int$$

The corresponding incrementing function can then be defined as

$$\mathit{incrementPair} :: \mathit{Pair} \to \mathit{Pair}$$
$$\mathit{incrementPair}\ (\mathit{MkPair}\ a\ b) = \mathit{MkPair}\ (a + 1)\ (b + 1)$$

incrementPair is pure boilerplate code: it simply follows the *shape* of *Pair* and increments every *Int* by one. This code also requires maintenance as *Pair* evolves. If we were to add a third element to the *Pair* type, we would have to update *incrementPair*, even though its high-level specification has not changed. Datatype-generic programming techniques solve this problem by allowing us to abstract over the *shape* of types, and define such operations once and for all. The approach used in the most famous solution, Scrap Your Boilerplate (SYB) [Lämmel and Peyton Jones, 2003], is to traverse the whole structure and perform runtime type comparisons to filter out the parts of interest. While this solution is simple and powerful, it is known to be slow due to the dynamic type tests. Instead, the generic-lens library performs a compile-time traversal over the shape of the data type, and generates optimal code that only accesses the pertinent parts at runtime. Using the interface provided by generic-lens, one can implement *incrementPair* by providing a one-to-one translation of the specification:

$$\mathit{incrementPair} :: \mathit{Pair} \to \mathit{Pair}$$
$$\mathit{incrementPair} = \mathit{over}\ (\mathit{types}\ \texttt{@}\mathit{Int})\ (+1)$$

That is, we simply instantiate the *types* function to focus on the *Int* values by using a visible type application. This generates a description of how to traverse all the *Int*s in the *Pair* data type. Finally, we apply the +1 function *over* the integers in the input *Pair*. This query is *type-directed*: users specify their type of interest as an input to the *types* function, and it returns a traversal scheme.

5.1.1 Algebraic structure

The *types* function worked, even though it has no knowledge about the *Pair* type. Indeed, *types* is defined generically over the *algebraic structure* of data types. Datatype-generic programming makes it possible to decompose data types into their constituent parts which are displayed in Figure 5.1. Data types in Haskell can be represented as sums of products. The sums (:+:) represent the choice between two constructors, and products (:×:) represent the fields inside a given constructor. For types with more than two constructors, the :+: type can be nested (similarly for products). A field of type a inside a constructor is marked as $K\ a$. Empty constructors (such as the empty list) are turned into U. Additionally, this generic representation contains metadata (names of data types, constructors, and

```
data f :+: g = L f | R g                 data M (m :: Meta) a = M a
data f :×: g = f :×: g                   data Meta = MetaData Symbol
data K a     = K a                                | MetaCons Symbol
data U       = U                                  | MetaSel (Maybe Symbol)
```

FIGURE 5.1: Sum-of-products encoding of algebraic data types

optionally field names) about the nodes. Every algebraic data type is isomorphic to a tree built out of these simple constructors. GHC can automatically generate this representation for most data types [Magalhães et al., 2010]. To get it to do so, we write the following line:

```
deriving instance Generic Pair
```

Finally, we can inspect what the sum-of-products tree representation of *Pair* looks like, by typing the following into GHCi:

```
> ghci> :kind Rep Pair
>  = M ('MetaData "Pair")
  (M ('MetaCons "MkPair")
    (M ('MetaSel 'Nothing) (K Int)
      :×: M ('MetaSel 'Nothing) (K Int)))
```

That is, *Pair* is a data type with a single constructor called *MkPair*, which contains two anonymous fields of type *Int*. Crucially, this information is available at the type-level. During compile-time, we can inspect the structure and generate optimised code.

5.1.2 Type-level traversals over the generic structure

The implementation of generic-lens defines several queries over the generic structure that is available at the type-level. For a comprehensive account of the implementation details, refer to Kiss et al. [2018]. Here, we present a selection of type families that are defined in the library. The exact details are not important, only to give an idea of what the current state of type-level programming in Haskell looks like.

```
type family HasCtor (ctor :: Symbol) f :: Bool where
  HasCtor ctor (M ('MetaCons ctor) _)
    = 'True
  HasCtor ctor (M _ f)
```

```
    = HasCtor ctor f
  HasCtor ctor (f :+: g)
    = HasCtor ctor f || HasCtor ctor g
  HasCtor ctor _
    = 'False
```

HasCtor ctor f recursively traverses the generic tree *f* and returns *'True* if the type contains a constructor named *ctor*, *'False* otherwise. It uses non-linear patterns (i.e. mentions the same name twice) to match on equal values. Another one is *HasField*:

```
type family HasField (field :: Symbol) f :: Bool where
  HasField field (M ('MetaSel ('Just field)) _)
    = 'True
  HasField field (M ('MetaSel _) _)
    = 'False
  HasField field (M _ f)
    = HasField field f
  HasField field (l :×: r)
    = HasField field l || HasField field r
  HasField field (l :+: r)
    = HasField field l || HasField field r
  HasField field (K _)
    = 'False
  HasField field U
    = 'False
```

HasField field f returns *'True* if and only if *f* contains a field named *field* (record types in Haskell have named fields). There are many more type families along these lines: they traverse the generic tree to extract some information.

Without focusing too much on the implementation details, we can see that both *HasCtor* and *HasField* are rather sizeable. After all, they handle all cases one-by-one, and recurse when appropriate. What's worse, they are almost identical, the only difference being the termination conditions. It is somewhat ironic that a library which was designed to eliminate boilerplate code itself contains a lot of boilerplate.

5.2 Type-level Generic programming

We now use `UnsaturatedTypeFamilies` to define a set of combinators that enable describing type-level traversals in a more concise manner. We then show how to implement the verbose type families from before as one-liners in the new scheme.

Recall the Scrap Your Boilerplate (SYB) library, which uses type equality tests to identify the relevant parts of data structures. We borrow this strategy and the interface for our type-level generic programming framework. Since we operate on types, the checks are of course going to happen at compile-time, rather than at runtime. We make use of *non-linear* patterns and *application decomposition* in order to traverse types generically.

Our first combinator is *Everywhere*, which takes a type function of kind $b \twoheadrightarrow b$, and applies it to every element of kind b in some structure st.

$$
\begin{aligned}
&\textbf{type}\ \textit{family}\ \textit{Everywhere}\ (f :: b \twoheadrightarrow b)\ (st :: a) :: a\ \textbf{where}\\
&\quad \textit{Everywhere}\ (f :: b \twoheadrightarrow b)\ (st :: b) \quad = f\ st\\
&\quad \textit{Everywhere}\ (f :: b \twoheadrightarrow b)\ (st\ x :: a) = (\textit{Everywhere}\ f\ st)\ (\textit{Everywhere}\ f\ x)\\
&\quad \textit{Everywhere}\ (f :: b \twoheadrightarrow b)\ (st :: a) \quad = st
\end{aligned}
$$

Everywhere makes use of several properties of closed type families [Eisenberg et al., 2014a], which we now make explicit.

Non-parametricity Type families can inspect members of any kind, even if the kind is polymorphic. That is, the first equation only matches when the domain of the function f is the same kind as the structure st. In this case, it applies the function and terminates.

Application decomposition The second pattern of the second equation looks like this: $st\ x$. It only matches when the input structure is some type constructor applied to some arguments. This means that it will match *Maybe Int*, but not *Int*, for example. In this case, *Everywhere* recurses into both sides of the application. We note that only *matchable* type constructor applications will match this pattern. This application decomposition in the type system is precisely the reason type families were not allowed unsaturated in the past.

Overlapping equations Finally, the third equation will match anything else. The patterns look exactly the same as in the first equation, except that this one does not require the kinds to match up. Since in this case, st is neither of the right kind nor is it an application that can be recursed into, we stop the traversal.

The second combinator, *Gmap*, is similar to *Everywhere* in that it applies a function to all elements of a given kind b in some structure of kind a. However, instead of leaving the new value in place in the structure, it returns the results in a list. As such, it can also change the kind of the elements into some result kind r.

$$
\begin{array}{l}
\textbf{type}\ \textit{family}\ \textit{Gmap}\ (f :: b \twoheadrightarrow r)\ (st :: a) :: [r]\ \textbf{where} \\
\quad \begin{array}{ll}
\textit{Gmap}\ f\ st & = \ '[f\ st] \\
\textit{Gmap}\ f\ (st\ x) & = \textit{Gmap}\ f\ st \mathbin{+\!\!+} \textit{Gmap}\ f\ x \\
\textit{Gmap}\ \_\ \_ & = \ '[\,]
\end{array}
\end{array}
$$

Both *Gmap* and *Everywhere* are *higher-order*: they take functions (in this case unsaturated type families) as arguments. Using *Gmap*, we define an auxiliary function *Listify* that collects all types of kind k into a list.

$$
\begin{array}{l}
\textbf{type}\ \textit{family}\ \textit{Listify}\ k\ (st :: a) :: [k]\ \textbf{where} \\
\quad \textit{Listify}\ k\ st = \textit{Gmap}\ (\textit{Id} :: k \twoheadrightarrow k)\ st
\end{array}
$$

Listify simply maps the identity function *Id*, but instantiated to the kind $k \twoheadrightarrow k$. This means that *Gmap* will only pick up the types whose kinds are k, and ignore the others in the resulting list. (Notice that k is given as an argument to *Listify*, and used in its return kind: indeed, the type system is dependently kinded, with the $\star :: \star$ axiom [Weirich et al., 2013].) For example, we can query all the names (types of kind *Symbol*) that appear in the definiton of the *Maybe* type by:

```
> ghci> :kind Listify Symbol (Rep (Maybe Int))
>  = '["Maybe","Nothing","Just"]
```

With our generic framework now in place, we can finally revisit the *HasField* and *HasCtor* functions.

$$
\begin{array}{l}
\textbf{type}\ \textit{family}\ \textit{HasCtor}_2\ \textit{ctor}\ f\ \textbf{where} \\
\quad \textit{HasCtor}_2\ \textit{ctor}\ f = \textit{Foldl}\ (||)\ \ '\textit{False}\ (\textit{Gmap}\ (==\ \ ('\textit{MetaCons}\ \textit{ctor}))\ f)
\end{array}
$$

We map the function $(==\ \ '\textit{MetaCons}\ \textit{ctor})$ over the structure. This implicitly selects only values of kind *Meta*, and returns $'\textit{True}$ for the constructors called *ctor*. We then fold the result with the $(||)$ function, defaulting to *False* in case the type had no constructors and *Gmap* returned an empty list. This results in $'\textit{True}$ if any of the constructors were called *ctor*. *Foldl* is simply the value-level *foldl* function lifted to the type-level (higher-order, so it was not possible to write before):

```
type family Foldl (f :: b ↠ a ↠ b) (z :: b) (xs :: [a]) :: b where
  Foldl f z '[] = z
  Foldl f z (x ': xs) = Foldl f (f z x) xs
```

Similarly, *HasField* can also be implemented as a one-liner type family:

```
type family HasField2 field f where
  HasField2 field f = Foldl (||) 'False (Gmap (== ('MetaSel ('Just field))) f)
```

We have seen how `UnsaturatedTypeFamilies` can help us eliminate boilerplate code that is typically present when writing type-level programs in Haskell. Indeed, we observe an *80%* decrease in the number of lines of the type-level code in the case of generic-lens. The resulting code is not only shorter, but also higher-level. It is now easier to see what the traversals do, and they remain correct even if the underlying generic representation were to be extended with new constructors.

5.3 Stateful programming

Handling state in a purely functional setting without mutation proves challenging. All of our functions have to take an extra parameter that encapsulates the local state of our program, and explicitly produce a new, potentially modified state. Worse, chaining these functions is tedious: we have to manually thread the state through the program by taking it from each stage of the pipeline and passing it to the next.

Wadler [1995] proposed *Monads* as the solution to this problem. Monads provide an abstraction for threading state through a pipeline of computations.

The interface in Haskell is defined as the following type class

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

With *return* taking a pure value and lifting it into a stateful one, and $\gg=$ (pronounced bind) passes the result of a stateful computation $m\ a$ to a continuation $a \to m\ b$ to ultimately produce a new stateful computation $m\ b$. Behind the scenes, it frees the programmer from the burden of passing state through.

Let us revisit this pattern and show that it is applicable in type-level computations. Promoting the *Monad* class to the type level is an exercise in syntactic rewriting, as type functions are now first class:

```
class Monad (m :: ⋆ → ⋆) where
  type Return (v :: a) :: m a
  type (>>=) (ma :: m a) (amb :: a ↠ m b) :: m b
```

Next, we define the *State s a* type, which represents a stateful computation that operates in some state of type s, and produces a value of type a. We intend to use *State* in a promoted context.

```
data State s a = State (s ↠ (s, a))
```

The auxiliary *RunState* function unwraps the function from *State*.

```
type family RunState (st :: State s a) :: (s ↠ (s, a)) where
  RunState ('State s) = s
```

Next, *ReturnState* is defined as a function that takes a value of type a, some state s, and wraps them up in a tuple.

```
type family ReturnState a s where
  ReturnState a s = '(s, a)
```

BindState uses a helper function, as we have no type-lambdas. We pass the state to the first computation, extract its result and state, then forward them to the continuation.

```
type family BindState (ma :: State s a) (amb :: a ↠ State s b) (st :: s) :: (s, b) where
  BindState ('State ma) f s = BindState' (ma s) (RunState . f)
type family BindState' (st :: (s, a)) (f :: a ↠ s ↠ (s, b)) :: (s, b) where
  BindState' '(s, a) f = f a s
```

Now we can write a *Monad* instance for the promoted *State* kind:

```
instance Monad (State s) where
  type Return a = 'State (ReturnState a)
  type (>>=) ma amb = 'State (BindState ma amb)
```

And the helper function ≫, which is a version of bind that discards the result of the first computation.

```
type (>>) ma f = ma >>= (Const f)
```

The *Modify* function takes some state transformer $st \twoheadrightarrow st$ and applies it to the ambient state.

```
type family Modify (f :: st ↠ st) :: State st () where
  Modify f = 'State (Modify' f)
type family Modify' (f :: st ↠ st) (s :: st) :: (st, ()) where
  Modify' f st = '(f st, '())
```

As a very simple stateful computation, we start with an initial state of 0, and modify the state 4 times by adding one each time.

```
type Stateful
  = Flip RunState 0
    $ Modify (Add 1)
    ≫ Modify (Add 1)
    ≫ Modify (Add 1)
    ≫ Modify (Add 1)
    ≫ (Return "Works")
```

Running the computation, we get

```
> ghci> :kind Stateful
>  = '(4, "Works")
```

Of course *Stateful* is a contrived example, but this technique can be useful for describing state machines at the type-level.

Chapter 6

Discussion

We have presented the `UnsaturatedTypeFamilies` GHC extension, which is based on the matchability distinction proposed by Eisenberg [2016]. However, we have shown how his subtyping relationship weakens type inference, and instead proposed *matchability polymorphism*.

We proposed type class instances for type families, a novel technique for ad-hoc overloading of functions, and specified an equational theory and restrictions under which instance resolution is decidable. Our equality relationship at the **class** role is conservative.

We also provide an implementation of `UnsaturatedTypeFamilies` in GHC, which is likely to be merged into the upstream compiler after a formal proposal process. The implementation is non-invasive, and well-specified in terms of GHC's constraint solving algorithm. As a demonstration of the robustness of our implementation, our fork of GHC can bootstrap itself, and all the examples in this work are valid type-checked programs.

We described a novel way of doing type-level generic programming by observing a unique interaction of Haskell's type system features that are not present in other dependently typed languages (namely non-parametricity and that application decomposition is possible on polymorphic type constructors).

6.1 Dependent Haskell

Dependent Haskell [Weirich et al., 2017] will blur the line between value-level and type-level programming, as arbitrary terms can then appear in types. Matchability is an important piece of the Dependent Haskell puzzle, and much of the development here can be repurposed in that context. Certain ergonomic features were not implemented as part of this project in anticipation of them becoming redundant in Dependent Haskell. Notably, type lambdas are not yet supported, but we discuss their interaction in the equational theory.

6.2 Future work

The type-level generic programming technique 5.2 is worth investigating more, in particular, because it requires application decomposition of polymorphic type constructors, which is likely to be removed from Dependent Haskell. The justification for the removal is that this feature is hard to reason about in the context of partiality, as we can observe an infinite number of inhabitants of any kind, which makes it impossible to use η-rules.

Bibliography

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Prog.*, 23, 2013.

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for Haskell. *J. Funct. Program.*, 26:1–79, 2016.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *International Conference on Functional Programming*, ICFP '05. ACM, 2005.

James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

Richard A. Eisenberg. System FC, as implemented in GHC. Technical Report MS-CIS-15-09, University of Pennsylvania, 2015. URL https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf.

Richard A Eisenberg. Dependent types in haskell: Theory and practice. *arXiv preprint arXiv:1610.07978*, 2016.

Richard A. Eisenberg and Jan Stolarek. Promoting functions to type families in Haskell. In *ACM SIGPLAN Haskell Symposium*, 2014.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 671–683, New York, NY, USA, 2014a. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535856. URL http://doi.acm.org/10.1145/2535838.2535856.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*, POPL '14. ACM, 2014b.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan Ahmed. Visible type application. In *European Symposium on Programming (ESOP)*, LNCS. Springer-Verlag, 2016.

Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15. ACM, 2015.

Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992. ISSN 0362-1340. doi: 10.1145/130697.130699. URL http://doi.acm.org/10.1145/130697.130699.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proc. 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107. ACM, 2004.

Csongor Kiss, Matthew Pickering, and Nicolas Wu. Generic deriving of generic traversals. In *International Conference on Functional Programming*, ICFP '18. ACM, 2018.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Types in Languages Design and Implementation*, pages 26–37, New York, NY, USA, 2003. ACM Press. doi: 10.1145/640136.604179.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863529. URL http://doi.acm.org/10.1145/1863523.1863529.

Sandy Maguire. `same-same-0.1.2` library, 2018. URL https://github.com/isovector/same-same.

Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. ISBN 88-7088-105-9. Notes by Giovanni Sambin.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Types in languages design and implementation*, TLDI '07. ACM, 2007.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5), September 2011.

Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-59451-2. doi: 10.1007/3-540-59451-5_2. URL http://dx.doi.org/10.1007/3-540-59451-5_2.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76. ACM, 1989.

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in haskell. *Proc. ACM Program. Lang.*, 1(ICFP):31:1–31:29, August 2017. ISSN 2475-1421. doi: 10.1145/3110275. URL http://doi.acm.org/10.1145/3110275.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, TLDI '12. ACM, 2012.