Higher-order Type-level Programming in Haskell

Csongor Kiss

Supervisor
Prof. Susan Eisenbach

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
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Implementation</td>
</tr>
<tr>
<td>4.1 Constraint-based type inference</td>
</tr>
<tr>
<td>4.1.1 Constraints</td>
</tr>
<tr>
<td>4.1.2 The solver pipeline</td>
</tr>
<tr>
<td>4.1.3 Unsaturated type families</td>
</tr>
<tr>
<td>5 Unsaturated type families in practice</td>
</tr>
<tr>
<td>5.1 Generic programming</td>
</tr>
<tr>
<td>5.1.1 Algebraic structure</td>
</tr>
<tr>
<td>5.1.2 Type-level traversals over the generic structure</td>
</tr>
<tr>
<td>5.2 Type-level Generic programming</td>
</tr>
<tr>
<td>5.3 Stateful programming</td>
</tr>
<tr>
<td>6 Discussion</td>
</tr>
<tr>
<td>6.1 Dependent Haskell</td>
</tr>
<tr>
<td>6.2 Future work</td>
</tr>
</tbody>
</table>
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:

```haskell
-- 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

\[
\text{saferHead} :: [a] \rightarrow \text{Maybe } a
\]

\[
\text{saferHead } (x : xs) = \text{Just } x
\]

\[
\text{saferHead } [] = \text{Nothing}
\]

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

\begin{verbatim}
> ghci> saferHead []
> Nothing
\end{verbatim}

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, [] :: [Int] and [1, 2] :: [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:

\[
\text{data Nat} = \text{Zero} \mid \text{Succ Nat}
\]

\(^2\)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.

```haskell
data Vector :: Nat → ⋆ → ⋆ where
  VNil :: Vector 'Zero a
  VCons :: a → Vector n a → Vector ('Succ n) a
```

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. 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 safeHead function as

```haskell
safeHead :: Vector ('Succ n) a → a
safeHead (VCons x xs) = x
```

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:

```haskell
> 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

---

3The ' (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.

\textbf{type family} Add \((a :: \text{Nat}) (b :: \text{Nat}) :: \text{Nat}\) \textbf{where}

\begin{align*}
\text{Add } '\text{Zero } b &= b \\
\text{Add } ('\text{Succ } a) b &= '\text{Succ } (\text{Add } a b)
\end{align*}

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

\textit{append} :: \text{Vector } n a \to \text{Vector } m a \to \text{Vector } (\text{Add } n m) a

\begin{align*}
\text{append } VNil v &= v \\
\text{append } (\text{VCons } a as) v &= \text{VCons } a (\text{append } as v)
\end{align*}

\section{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 \textit{map} function in Haskell:

\begin{align*}
\text{map} :: (a \to b) \to [a] \to [b] \\
\text{map} - [] &= [] \\
\text{map } f (x : xs) &= f x : \text{map } f xs
\end{align*}

\textit{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.

\begin{verbatim}
> ghci> map (1+) [1, 2, 3, 4]
> [2, 3, 4, 5]
\end{verbatim}
Here, \((1+)\) is a partially applied function: we have provided a single argument to \((+):\mathbb{Int} \to \mathbb{Int}\), resulting in a new function of type \(\mathbb{Int} \to \mathbb{Int}\) that takes only one argument. We could try and define the type-level equivalent of \texttt{map} as follows:

\[
\text{type family Map } (f :: a \to b) \ (xs :: [a]) :: [b] \text{ where}
\]
\[
\begin{align*}
\text{Map } \_ \_ \_ [\_] &= [\_] \\
\text{Map } f \ (x \ : \ xs) &= f \ x \ : \ \text{Map } f \ xs
\end{align*}
\]

\(\text{Map}\) operates on the promoted list type (hence the use of the \('\) symbol in the constructors). While this definition of \texttt{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 following expression is rejected:

\[
\text{> ghci> } \text{kind } \text{Map } (\text{Add } (\text{‘Succ } \text{‘Zero})) \ [\text{‘Zero}]
\]

where \texttt{Add} is the type family defined above. The problem here is that \texttt{Add} is given a single argument, but its arity is 2. This means that in \(\text{Add } (\text{‘Succ } \text{‘Zero})\), \texttt{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 \texttt{Map} to every possible use-case:

\[
\text{type family MapAddOne } (xs :: [\text{Nat}]) :: [\text{Nat}] \text{ where}
\]
\[
\begin{align*}
\text{MapAddOne } \_ \_ \_ [\_] &= [\_] \\
\text{MapAddOne } (x \ : \ xs) &= \text{‘Succ } x \ : \ \text{MapAddOne } xs
\end{align*}
\]

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 \texttt{even :: Int \rightarrow Bool} function to an argument whose type is \texttt{String}, as a \texttt{String} value can’t be used where an \texttt{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 \texttt{Int Int} (\texttt{Int} applied to itself) is clearly bogus, as the \texttt{Int} type takes no arguments. An example of a type that does take an argument is \texttt{Maybe}:

\begin{verbatim}
data Maybe a = Nothing | Just a
\end{verbatim}

\texttt{Maybe} is a \textit{type constructor}: it takes a type argument and returns a new type. The \texttt{Maybe Int} type is the result of applying \texttt{Maybe} to \texttt{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 \texttt{Int}, \texttt{String}, etc.) have kind \(\star\) (pronounced \texttt{star}, or simply \texttt{type}). Thus the kind of \texttt{Maybe} is \(\star \rightarrow \star\). The \texttt{Maybe} type constructor abstracts over types, but there are ways of abstracting over type constructors too. As an example, consider the \textit{Functor} type class from the Haskell standard library:

```haskell
class Functor f where
    fmap :: (a \rightarrow b) \rightarrow f a \rightarrow f b
```

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

```haskell
instance Functor Maybe where
    fmap :: (a \rightarrow b) \rightarrow Maybe a \rightarrow Maybe b
    fmap f Nothing = Nothing
    fmap f (Just a) = Just (f a)
```

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

```haskell
class Functor (f :: \star \rightarrow \star) where
    fmap :: (a \rightarrow b) \rightarrow f a \rightarrow f b
```

The base kind system in Haskell 98 \cite{Peyton98} consists only of two formation rules:

\[
\kappa ::= \star \mid \kappa \rightarrow \kappa
\]

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 \textit{open}: new inhabitants can be added at any time simply by creating new data types:

```haskell
data Wurble
```

We get a new type \texttt{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:

\[
\begin{align*}
\text{data } & \ HZero \\
\text{data } & \ H\text{Succ } n
\end{align*}
\]

They specify that \( HZero \) means zero, and \( H\text{Succ } n \) is the successor of \( n \). For instance, the number two is represented as \( H\text{Succ } (H\text{Succ } HZero) \). 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 \( H\text{Succ} \) is \( \star \rightarrow \star \), which means that as far as the type checker is concerned, \( H\text{Succ } \text{Bool} \) 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 \( \text{Nat} \) type from 1.2:

\[
\begin{align*}
\text{data } & \ \text{Nat} \\
& = \ Zero \\
& \mid \ Succ \ \text{Nat}
\end{align*}
\]

\( \text{Nat} \) extends the set of kinds, and has two inhabitants, \( \text{Zero} :: \text{Nat} \) and \( \text{Succ} :: \text{Nat} \rightarrow \text{Nat} \). In this new scheme, the type \( \text{Succ} \ \text{Bool} \) is ill-kinded, as \( \text{Bool} \) has kind \( \star \) instead of \( \text{Nat} \). 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 ask the question: how are kinds classified? What’s the “type” of \( \text{Nat} \), 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: \( \mathcal{U}_0 \in \mathcal{U}_1 \in \mathcal{U}_2 \in \ldots \)

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 \( \text{Nat} :: \star \), and \( \star :: \star :: \star :: \ldots \). 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 syntactic equivalence, or being related by a type synonym:

```haskell
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:

```haskell
type family Elem c

class Collection c where
  empty :: c
  insert :: Elem c \to c \to 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:

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

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 \(Chars\).

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

\begin{verbatim}
> ghci> :type insert 'c'
> insert 'c' :: (Collection c, Elem c ~ Char) \Rightarrow c \rightarrow c
\end{verbatim}

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

\begin{verbatim}
> ghci> :type insert @String
> insert @String :: Char \rightarrow String \rightarrow String
\end{verbatim}

\cite{Eisenberg2016} introduced \textit{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. \cite{Eisenberg2014} extended type families with the ability to define a closed set of overlapping equations. To illustrate this, we define the following type family:

\begin{verbatim}
type family Equals (a :: k) (b :: k) :: Bool where
Equals a a = 'True
Equals a b = 'False
\end{verbatim}

\(\text{Equals}\) 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:

\[
\text{type family Add} \ (a :: \text{Nat}) \ (b :: \text{Nat}) :: \text{Nat where}
\]

\[
\text{Add} \ '\text{Zero} b = b
\]

\[
\text{Add} \ ('\text{Succ} \ a) b = '\text{Succ} (\text{Add} \ a b)
\]

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:

\[
\text{Add} \ '\text{Zero} b \sim b
\]

\[
\text{Add} \ ('\text{Succ} \ a) b \sim '\text{Succ} (\text{Add} \ a b)
\]

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

\[
> \text{ghci} > \text{kind} (\text{Add} ('\text{Succ} (\text{Zero})) ('\text{Succ} (\text{Zero})))
\]

\[
> = '\text{Succ} ('\text{Succ} '\text{Zero})
\]

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:

\[
\text{type family Bottom where}
\]

\[
\text{Bottom} = \text{Bottom}
\]

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.
Chapter 2. Background

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

\[
\text{replaceWith10} :: \text{Functor } f \Rightarrow f a \rightarrow f \text{ Int}
\]

\[
\text{replaceWith10 } fa = \text{fmap } (\_ \rightarrow 10) \ fa
\]

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 \( f a \sim \text{Maybe Bool} \). The type checker proceeds by \textit{decomposing} this equality into two smaller equalities, \( f \sim \text{Maybe} \) and \( a \sim \text{Bool} \). These can now be turned into a substitution map \([f \mapsto \text{Maybe}, a \mapsto \text{Bool}]\), and type inference is done: the type of replaceWith10 here is Maybe Bool \( \rightarrow \) Maybe Int. The way we could proceed was by taking apart the application \( f a \).

Now consider an example with hand-written equalities:

\[
\text{decompose} :: (f a \sim g b) \Rightarrow a \rightarrow b
\]

\[
\text{decompose} = \text{id}
\]

The type of decompose is \( a \rightarrow b \), but it is implemented with \text{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 \( a \rightarrow a \), which is now of course compatible with the implementation \text{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.

\[
\text{type family AlwaysInt a where}
\]

\[
\text{AlwaysInt } _ = \text{Int}
\]

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:
Chapter 2. Background

bad :: Integer → Char
bad = decompose @AlwaysInt @Integer @AlwaysInt @Char

Type type of bad is \((\text{AlwaysInt} \text{ Integer} \sim \text{AlwaysInt} \text{ Char}) \Rightarrow \text{Integer} \rightarrow \text{Char}\). The equality constraint normalises to \(\text{Int} \sim \text{Int}\) (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 $\text{Const}$ function and the identity $\text{Id}$ can be represented as data types:

```haskell
type TyFun a b = a \to b \to *
infixr 0 'TyFun'
data SymConst2 ::  * '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. $\text{Apply}$ dispatches on the defunctionalisation symbols:

```haskell
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 $\text{SymConst2}$ case. Then we can write a version of $\text{Map}$ that takes a defunctionalisation symbol, and applies it to every element of a list with $\text{Apply}$.

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

```
> 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 $\text{Apply}$ function to handle it appropriately. Indeed, the $\text{singletons}$ 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 $\text{TyFun}$ and $\text{Apply}$ into the type system, but without doing defunctionalisation. The important property of $\text{TyFun}$ is that is syntactically a different arrow from $\to$, 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 \( \rightarrow \), and the arrow of type functions (unmatchable) is \( \Rightarrow \).

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 \( \text{Map} \) type family from Section 1.4, but now with an unmatchable function as its argument:

\[
\text{type family Map} \ (f :: a \rightarrow b) \ (xs :: [a]) :: [b] \ \text{where}
\]

\[
\begin{align*}
\text{Map} & \quad \text{[]} = \quad \text{[]} \\
\text{Map} \ f \ (x \ : \ xs) & = f \ x \ : \ \text{Map} \ f \ xs
\end{align*}
\]

\( \text{Map} \) can now be called with a partially applied type family, as its argument’s kind suggests\(^1\):

\[
> \text{ghci} > \text{kind Map (Add ('}\text{Succ} \ '\text{Zero}) \ ')\text{[('}\text{Succ} \ '\text{Zero}), 'Zero]}
\]

\[
> = \text{'}\text{[('}\text{Succ ('}\text{Succ} \ '\text{Zero}}), '\text{Succ Zero}]
\]

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 \texttt{Lhs2TeX} tool.
data Person f = MkPerson
    { name :: f @@ String,
      age :: f @@ Int
    }

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

\[
\text{person} :: \text{Person Id}
\]
\[
\text{person} = \text{MkPerson "Toby" 21}
\]

The real utility of this parameterisation of course comes from plugging in more interesting functions. For example, suppose that we want to serialise the \textit{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:

\[
\text{type family DbPrimitive a}
\]
\[
\text{type instance DbPrimitive String } = \text{DbText}
\]
\[
\text{type instance DbPrimitive Int } = \text{DbInt}
\]

We could then write a function with the signature

\[
\text{serialise} :: \text{Person Id } \rightarrow \text{Person DbPrimitive}
\]
\[
\text{serialise} \ (\text{MkPerson name age}) = \text{MkPerson (mkDbText name) (mkDbInt age)}
\]

Or perhaps a pretty-printable version of \textit{Person}, that contains only \textit{String} values:

\[
\text{pretty} :: \text{Person Id } \rightarrow \text{Person (Const String)}
\]
\[
\text{pretty} \ (\text{MkPerson name age}) = \text{MkPerson (show name) (show age)}
\]

Where \textit{Const} is the constant function:

\[
\text{type family Const a b where}
\]
\[
\text{Const a b } = a
\]

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 \textit{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 Ints. As the signature of addOne requires that the input is of the form f Int, we cannot 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 \rightarrow 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:

\[
\text{addOne} \text{Int} x = \text{unwrap} (\text{addOne} (\text{Identity} x))
\]

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 Ints. 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:

\[
\begin{align*}
\text{instance Functor}' (f :: \star \rightarrow \star) \text{ where } \\
\text{fmap}' :: (a \rightarrow b) \rightarrow f \odot a \rightarrow f \odot b
\end{align*}
\]

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 \( \star \rightarrow \star \)), but Id has kind \( \star \rightarrow \star \).

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

\[
\text{instance } \text{Functor}' \ \text{Id} \ \text{where} \\
\quad \text{fmap}' :: (a \to b) \to a \to b \\
\quad \text{fmap}' \ f \ x = f \ x
\]

But how would a call to \( \text{fmap}' \) 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 \((\text{show} :: \text{Show } a \Rightarrow a \to \text{String})\) on a value of type \( \text{Int} \), we instantiate the constraint to \( \text{Show } \text{Int} \) and find an existing instance in scope. Notice, however, that the type of \( \text{fmap}' \) is ambiguous in the \( \text{Functor}' \ \text{Id} \) instance: \( \text{Id} \) itself is not manifest. To demonstrate the problem, consider the following use of \( \text{fmap}' \):

\[
\text{fmap}' \ (+1) \ 10 :: \text{Int}
\]

There is no way to determine which instance of \( \text{Functor}' \) was used to produce the result. From \( \text{fmap}' \)'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 \( \text{Int} \to \text{Int} \) we can deduce that \( a \) must be \( \text{Int} \). Thus we’re left with the problem of solving \( f @ \text{Id} \text{Int} \sim \text{Int} \). That is, given that some function \( f \) applied to \( \text{Int} \) results in \( \text{Int} \), what is \( f \)?

To address instance resolution at the use sites, we recall the problematic application from before: \( \text{fmap}' \ (+1) \ 10 :: \text{Int} \). 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: \( \text{fmap}' \ @\text{Id} \ (+1) \ 10 :: \text{Int} \). Here, the constraint solver starts off with a given equality of \( f \sim \text{Id} \) as it tries to solve \( f @\text{Id} \text{Int} \sim \text{Int} \), which can now happily proceed by substituting \( \text{Id} \) 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 \( \text{Functor}' \) class: modulo matchability, it is defined in the same way as the original \( \text{Functor} \). We discuss two alternatives for eliminating this code duplication.
3.3.1 Subtyping

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

\[
\text{fmap}'\ (+1) \ x :: \text{Maybe} \ \text{Int}
\]

What’s the type of \( x \)? From the signature of \( \text{fmap}' \) we can conclude that it’s of shape \( f \oplus a \). Furthermore, \((+1)\) has type \( \text{Int} \rightarrow \text{Int} \), therefore \( a \) is \( \text{Int} \). Similarly to before, we’re faced with the problem of \( f \oplus \text{Int} \sim \text{Maybe} \ \text{Int} \), with the unknown \( f \). Since \( f \)’s kind is \( \star \rightarrow \star \), 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 (\( \text{Functor} \)) and code reuse (\( \text{Functor}' \)).

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 \( \text{ARROW} \) that is parameterised over its matchability.

\[
\text{data} \ \text{Matchability} = \text{Matchable} \mid \text{Unmatchable}
\]
\[
\text{type} \ \text{ARROW} :: \text{Matchability} \rightarrow \star \rightarrow \star \rightarrow \star
\]

Then the two arrows \( \rightarrow \) and \( \rightarrow^\star \) simply become synonyms for the two possible instantiations of \( \text{ARROW} \):

\[
\text{type} \ (\rightarrow) = \text{ARROW} \ ' \text{Matchable}
\]
\[
\text{type} \ (\rightarrow^\star) = \text{ARROW} \ ' \text{Unmatchable}
\]
\[
\text{type} \ (\oplus) :: \text{Arrow} \ m \ k \ j \rightarrow \text{Arrow} \ m \ k \ j
\]

This allows us to define \( \text{FunctorP} \) class, which is the matchability-polymorphic version of \( \text{Functor} \):

\[
\text{class} \ \text{FunctorP} \ (f :: \text{ARROW} \ m \star \star) \ \text{where}
\]
\[
\text{fmapP} :: (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b
\]

Then we can instantiate \( \text{FunctorP} \) 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 \( x \):

\[ f \text{mapP} (+1) \ x :: \text{Maybe Int} \]

From the signature of \( f \text{mapP} \) we gather that \( x \)'s type is of shape \( f \circ a \), and from \((+1) :: \text{Int} \to \text{Int}\) that \( a \sim \text{Int} \). Thus the type inference problem is \( f \circ a \sim \text{Maybe Int} \). As before, we're stuck at this point. However, the kind of \( f \) is \( \text{Arrow} \ m \star \star \) 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 \( f \text{Int} \sim \text{Maybe Int} \), which can now be decomposed over the applications to get the wanted constraint \( f \sim \text{Maybe} \), which can be solved with unification, arriving the the solution \( x :: \text{Maybe Int} \).

This means that \( f \text{mapP} \) behaves exactly like \( f \text{map} \) (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:

\[ f \text{mapP} \circ \text{Id} (+1) \ x :: \text{Int} \]

Here, the constraint is once again \( f \circ \text{Int} \sim \text{Int} \), but with a given constraint that \( f \sim \text{Id} \). 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 \( \text{Int} \).

## 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
Chapter 3. Unsaturated type families

Metavariables:
- $x$ term
- $a, b$ type
- $c$ coercion
- $C$ axiom
- $F$ type family
- $m$ matchability

\[
\begin{align*}
  e & ::= \lambda x : \tau.e \mid \Lambda a : \kappa.e \mid \lambda c : \phi.e \mid e \, \gamma \mid e_1 \, e_2 \mid e \, \tau \mid e \triangleright \gamma & \text{terms} \\
  \tau, \sigma & ::= a \mid \tau_1 \otimes^\mu \tau_2 \mid \forall a : \kappa.\tau \mid H & \text{types} \\
  \kappa & ::= \star \mid \kappa_1 \to^\mu \kappa_2 & \text{kinds} \\
  H & ::= (\to) \mid (\Rightarrow) \mid (\sim^\kappa) \mid F & \text{type constants} \\
  \phi & ::= \tau \sim^\kappa \sigma & \text{propositions} \\
  \Phi & ::= [a : \kappa].\tau \sim \sigma & \text{axioms} \\
  \gamma, \eta & ::= \text{coercions} \\
  \mu, \nu & ::= m \mid M \mid U & \text{matchability} \\
  \Gamma & ::= \emptyset \mid \Gamma, a : \kappa \mid \Gamma, c : \phi \mid \Gamma, x : \tau \mid \Gamma, m : \mu & \text{typing contexts}
\end{align*}
\]

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 $\tau$ and $\sigma$ at kind $\kappa$. Coercions can be abstracted over with $\lambda c : \phi.e$ and applied as $e \, \gamma$. For example, the function

$$
\text{co} :: (a \sim \text{Int}) \Rightarrow a \to \text{Int}
$$

$$
\text{co} = \text{id}
$$

abstracts over a proof that $a \sim \text{Int}$. 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 \( \gamma \).

### 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 \( \tau_1 \) into type \( \tau_2 \) is by providing a witness (a coercion) \( \gamma \) of the type equality \( \tau_1 \sim \tau_2 \) and explicitly casting \( e \) by \( \gamma \): \( e \triangleright \gamma \). The corresponding typing rule is \( E_{\text{Cast}} \) 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 (\( \text{Co.Refl} \)), symmetry (\( \text{Co.Sym} \)), and transitivity (\( \text{Co.Trans} \)) rules) with congruence. This way, type checking in \( FC \) is syntactic, as all the derivations are encoded in the terms via casts. For example, in order to use a coercion \( \gamma : \text{Bool} \sim a \) to prove that \( e : a \) has type \( e : \text{Bool} \), we cast \( e \) as such: \( e \triangleright \text{sym} \gamma \). 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 \( FC \) by parameterising the arrow kind and the type application form with their matchability. Correspondingly, the coercion decomposition rules \( \text{Co.Left} \) and \( \text{Co.Right} \) are only allowed to decompose matchable applications. Notably, the \( Ty_{\text{App}} \) kinding rule allows the application form to be reused for both kind of matchabilities. Rule \( Ty_{\text{Fun}} \) assigns an unmatchable arrow kind to type functions.
Chapter 3. Unsaturated type families

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
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 Id_2 a where
  Id_2 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' Id_2 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
Figure 3.5: Formation rules for coercions
not answer these questions until we can make a decision on whether or not \( \text{Id} \) and \( \text{Id}_2 \) 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 \( \alpha \)-equivalence). The normalisation steps are the usual \( \beta \) and \( \eta \) reductions from pure lambda calculus, extended with \( \delta \)-reduction, which corresponds to expansion of constants, such as type family equations in our case. We can show that \( \text{Id} \) and \( \text{Id}_2 \) are \( \beta\delta\eta \)-convertible:

\[
\text{Id} \equiv \eta \lambda x. \text{Id} \ x \\
\equiv \delta \lambda x.x
\]

\[
\text{Id}_2 \equiv \eta \lambda y. \text{Id}_2 \ y \\
\equiv \delta \lambda y.y
\]

\[
\lambda x.x \equiv \alpha \lambda y.y
\]

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 ($\beta$). As of writing, it has no $\eta$ rules, but there is a real possibility of adding them. This would mean that $\text{Id}_2$ and $\text{Id}$ 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 $\text{Bool} \sim_N \text{Bool}$), or there is a nominal equality axiom relating them. Such axioms are introduced by type family equations. For example, consider the type family

\[
\text{type family AlwaysBool a where}
\]

\[
\text{AlwaysBool } \_ = \text{Bool}
\]

$\text{AlwaysBool}$ introduces an axiom scheme that for all $a$, $\text{AlwaysBool } a \sim_N \text{Bool}$. These axioms are used in $\delta$-reduction when equating two types. In today’s Haskell,
nominal equality is the same as \( \delta \)-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 \( Age \sim_R \text{Int} \). However, note that \( Age \not\sim_N \text{Int} \) (this
distinction is the reason for using newtypes, 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 \( \text{Int} \) values into a list of \( Age \) values. We could write \( \text{map Age xs} \), 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 \( \text{Int} \sim_R Age \) equality to \([\text{Int}] \sim_R [Age]\) by congruence over type applications.

Phantom equality Any two types are equal at the phantom role \((\text{Int} \sim_P \text{Bool})\). 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,
\( \text{Phantom Bool} \sim_R \text{Phantom String} \), despite \( \text{Bool} \not\sim_R \text{String} \) (but \( \text{Bool} \sim_P \text{String} \)).

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.
Chapter 3. Unsaturated type families

Since we have no type lambdas as of yet (so $\beta$-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 ($\delta$). 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, $\text{Maybe} \,(\text{Id} \,\text{Int})$ steps to $\text{Maybe} \,\text{Int}$ via Red_App and Red_Axiom. In the latter, we find that there is an axiom $\text{Id} \,a \sim a$, and that $\text{Id} \,\text{Int}$ can be obtained from $(\text{Id} \,a)[a \mapsto \text{Int}]$. 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 $\tau$ 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, \( f \otimes a \) 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, \( f : k \rightarrow j \rightarrow l \) for some kinds \( k, j, \) and \( l \).

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

```haskell
class Bad (a :: ⋆ → ⋆)
instance Bad (f @@ Int)
instance Bad (f @@ Char)
```

In both instances, we are matching on \( f :: ⋆ → ⋆ → ⋆ \) applied to some type. There is no guarantee that a type function of kind \( ⋆ → ⋆ → ⋆ \) 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):

```haskell
type family F (a :: ⋆) :: ⋆ → ⋆ where
    F Int = G Char
type family G (a :: ⋆) :: ⋆ → ⋆ where
    G Char = F Int
```

In the first instance of \( \text{Bad} \), instantiating \( f \) to \( F \) would mean an instance lookup for \( \text{Bar} (F @@ Int) \). This either matches the first instance, or reduces to \( \text{Bar} (G @@ Char) \) 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. \( \square \)

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

**Proof.** By induction on the structure of \( \tau \).

**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 \texttt{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 \texttt{Red.Refl} (as variables do not appear on the left hand side of axioms).

**Case** \( \tau = \tau_1 \emptyset \mu \tau_2 \): Case split on \( \mu \):

**Case M:** Axioms only have unmatchable applications on the left-hand side, so the only non-trivial step is by \texttt{Red.App}. By inversion on \texttt{Red.App} we get that \( \sigma_1 = \sigma_{11} \emptyset^M \sigma_{12} \) and \( \sigma_2 = \sigma_{21} \emptyset^M \sigma_{22} \) 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 \( \tau \) is an under-saturated type family application, then the only way for it to step is via the \texttt{Red.App} rule. The proof then follows as in the previous case. We prove the saturated case by induction on the two reductions.

**Case \texttt{Red.App}/\texttt{Red.App}:** As above.

**Case \texttt{Red.App}/\texttt{Red.Axiom}:** We know that \( \tau_1 \rightsquigarrow \sigma_{11} \) and \( \tau_2 \rightsquigarrow \sigma_{12} \) by inversion on \texttt{Red.App}. The other case reduced with an axiom, so \( \tau = F(\bar{\pi}) \) for some type family \( F \). Let the matching axiom be \( C : [\bar{a} : \bar{\pi}] F(\bar{a}) \sim \sigma' \). Then \( \sigma_2 = \sigma'[\bar{\pi}_0/\bar{a}] \). Since \( \tau = F(\bar{\pi}) \), \( \sigma_{11} \) 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 \( \sigma_2 \) did. Then we appeal to the substitution lemma, and we’re done.

**Case \texttt{Red.Axiom}/\texttt{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).

\[
\textbf{Theorem} \text{ (Instance head selection is decidable).} \quad \text{Given a type } \tau \text{ such that } \tau \rightsquigarrow^* \sigma, \text{ and } \sigma \text{ is in normal form, and a class } C \text{ a whose instance heads are all valid, instance selection for } \tau \text{ is decidable.}
\]

\textit{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: $\xi$ type function-free types

\[
\begin{align*}
Q & ::= \epsilon \mid Q_1 \land Q_2 \mid \tau_1 \sim \tau_2 & \text{constraints} \\
Q & ::= Q \mid Q_1 \land Q_2 \mid \forall \vec{a}. F \xi \sim \tau & \text{top-level axiom schemes} \\
l & ::= g \mid w & \text{constraint flavours}
\end{align*}
\]

**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 $\Gamma$, $e$ has type $\tau$. For example, the following function is checked with the given constraint $a \sim \text{String}, b \sim \text{Int}$:

\[
\begin{align*}
\textit{lengthy} :: (a \sim \text{String}, b \sim \text{Int}) & \Rightarrow a \rightarrow b \\
\textit{lengthy} = \text{length}
\end{align*}
\]

the corresponding judgement is

\[
(a \sim \text{String}, b \sim \text{Int}); (\text{length} : \text{String} \rightarrow \text{Int}) \vdash \text{length} : a \rightarrow b
\]

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

\[
\Gamma \mapsto e : \tau \sim Q_w
\]

That is, we infer that $e$ has type $\tau$ in the type environment $\Gamma$, and we generate constraint $Q_w$.

### 4.1.2 The solver pipeline

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

\[
Q ; Q_g \xrightarrow{s\text{imp}} Q_w \sim Q_r ; \theta
\]

That is, it takes the top level set of axiom schemes ($Q$), the set of given constraints ($Q_g$), and the set of wanted constraints ($Q_w$). It then produces a type substitution map $\theta$, and a set of residual constraints ($Q_r$): these are constraints that could not be simplified further. If $Q_r$ is $\epsilon$ (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:
weirdId :: Id a ~ b ⇒ a → b
weirdId x = x

Here, we are given (from the context) that Id a ~ b, and the generated wanted is a ~ b (from the function’s definition).

\[
\begin{align*}
[g] & \quad \text{Id } a \sim b \\
[w] & \quad a \sim b
\end{align*}
\]

First we process the givens, and find that Id a can be reduced to a, and our given constraint becomes a ~ 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 ~ Id a, using it as a rewrite rule will result in continuous expansion of as into Id as 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 ~ Bool and a ~ b interact, as we can learn that b ~ 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:
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:

- \( F \tau \prec \tau \) when \( \tau \neq G \tau \)
- \( tv_1 \prec tv_2 \) when \( tv_1 \leq tv_2 \) lexicographically
- \( tv \prec \xi \)

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.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFL</td>
<td>( \text{canon}[l] (\tau \sim \tau) = {\epsilon, \epsilon} )</td>
<td></td>
</tr>
<tr>
<td>APPDEC</td>
<td>( \text{canon}[l] (\tau_1 \oplus^M \tau_2 \sim \tau_3 \oplus^M \tau_4) = {\epsilon, \tau_1 \sim \tau_2 \land \tau_3 \sim \tau_4} )</td>
<td></td>
</tr>
<tr>
<td>OCCCHECK</td>
<td>( \text{canon}[l] (tv \sim \xi) ) when ( tv \in \xi, \xi \neq tv ) = (\perp)</td>
<td></td>
</tr>
<tr>
<td>ORIENT</td>
<td>( \text{canon}[l] (\tau_1 \sim \tau_2) ) where ( \tau_1 \prec \tau_2 ) = ({\epsilon, \tau_2 \sim \tau_1})</td>
<td></td>
</tr>
<tr>
<td>FFLATWL</td>
<td>( \text{canon}[w] (F[G\xi] \sim \tau) = {\epsilon, (F[\beta] \sim \tau) \land (G\xi \sim \beta)} )</td>
<td></td>
</tr>
<tr>
<td>FFLATGL</td>
<td>( \text{canon}[g] (F[G\xi] \sim \tau) = {[\beta \mapsto G\xi], (F[\beta] \sim \tau) \land (G\xi \sim \beta)} )</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Canonicalisation rules
The canonicaliser $\text{canon}[l]$ produces a flattening substitution (see below), and a set of new constraints, whose flavour is $l$.

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

**Flattening**

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

**Canonicalising equalities**

This is the stage where equalities are decomposed, by rule $\text{APPDEC}$. 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 $\text{canon}[l]$ 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 $\text{CFEQ}$ 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, $\vdash_{\text{can}} \xi_1 \#_{\#} \xi_2 \sim \xi_3$. 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 $\text{CFEQ}$ rule is that the type family is known.

We define $\text{Apply}$ as a special type family, whose first argument is an unknown unsaturated type family. Applications are of the form $\text{Apply} f \xi$. When the type family becomes
known to be some $F$, the special form is $\text{Apply } F \, \xi$ is rewritten to $F \, \xi$. The only change we need to make is an extra flattening rule which replaces unmatchable applications to type variables with the $\text{Apply}$ family, and another canonicalisation rule that replaces applications of $\text{Apply}$ 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:

\[
\text{data } \text{Pair} = \text{MkPair Int Int}
\]

The corresponding incrementing function can then be defined as
incrementPair :: Pair → Pair
incrementPair (MkPair a b) = 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:

incrementPair :: Pair → Pair
incrementPair = over (types @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 Ints 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
Chapter 5. Unsaturated type families in practice

46

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.

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.

\[
\begin{align*}
data f :+ g &= L f | R g \\
data f :\times g &= f :\times g \\
data K a &= K a \\
data U &= U \\
data M (m :: Meta) a &= M a \\
data Meta &= \text{MetaData Symbol} \\
&\quad | \text{MetaCons Symbol} \\
&\quad | \text{MetaSel (Maybe Symbol)} \\
\end{align*}
\]

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:

\[ \text{deriving instance Generic Pair} \]

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

\[
\text{ghci}\> :\text{kind Rep Pair}\n\]

\[
\begin{align*}
\text{=} &= M (\text{"MetaData "Pair"}) \\
&= (M (\text{"MetaCons "MkPair"}) \\
&\quad (M (\text{"MetaSel "Nothing}) (K \text{Int}) \\
&\quad :\times M (\text{"MetaSel "Nothing}) (K \text{Int})))
\end{align*}
\]

That is, \texttt{Pair} is a data type with a single constructor called \texttt{MkPair}, which contains two anonymous fields of type \texttt{Int}. Crucially, this information is available at the type-level. During compile-time, we can inspect the structure and generate optimised code.
\[
\text{HasCtor } \text{ctor } \text{f} \\
\text{HasCtor } \text{ctor } (\text{f} \downarrow: \text{g}) \\
= \text{HasCtor } \text{ctor } \text{f} \parallel \text{HasCtor } \text{ctor } \text{g} \\
\text{HasCtor } \text{ctor } _- \\
= 'False
\]

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

\textbf{type family HasField } (\text{field} :: \text{Symbol}) \text{ f :: } \text{Bool where} \\
\text{HasField } \text{field} \ (M ('\text{MetaSel} ('\text{Just field})) _) \\
= 'True \\
\text{HasField } \text{field} \ (M ('\text{MetaSel} _) _) \\
= 'False \\
\text{HasField } \text{field} \ (M _f) \\
= \text{HasField } \text{field} \text{ f} \\
\text{HasField } \text{field} \ (l :\times: r) \\
= \text{HasField } \text{field} \ l \parallel \text{HasField } \text{field} \ r \\
\text{HasField } \text{field} \ (l :\downarrow: r) \\
= \text{HasField } \text{field} \ l \parallel \text{HasField } \text{field} \ r \\
\text{HasField } \text{field} \ (K _) \\
= 'False \\
\text{HasField } \text{field} \ U \\
= 'False
\]

\text{HasField } \text{field} \ f \text{ returns } 'True \text{ if and only if } \text{f} \text{ contains a field named } \text{field} \text{ (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 \text{HasCtor} \text{ and } \text{HasField} \text{ 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.}
Chapter 5. Unsaturated type families in practice

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 \( b \to b \), and applies it to every element of kind \( b \) in some structure \( st \).

\[
\text{type family Everywhere (} f :: b \to b \text{) (} st :: a \text{) :: a where}
\]

\[
\begin{align*}
\text{Everywhere (} f :: b \to b \text{) (} st :: b \text{)} & = f \ st \\
\text{Everywhere (} f :: b \to b \text{) (} st x :: a \text{)} & = (\text{Everywhere } f \ st) (\text{Everywhere } f \ x) \\
\text{Everywhere (} f :: b \to b \text{) (} st :: a \text{)} & = st
\end{align*}
\]

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, \textit{Gmap}, is similar to \textit{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{verbatim}

\textbf{type family Gmap} (f :: b \rightarrow r) (st :: a) :: [r] \textbf{where}
Gmap f st       = '[f st]
Gmap f (st x)   = Gmap f st ++ Gmap f x
Gmap _         = '[]
\end{verbatim}

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

\begin{verbatim}

\textbf{type family Listify} k (st :: a) :: [k] \textbf{where}
Listify k st    = Gmap (Id :: k \rightarrow k) st
\end{verbatim}

\textit{Listify} simply maps the identity function \( \text{Id} \), but instantiated to the kind \( k \rightarrow k \). This means that \textit{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 \textit{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 \textit{Symbol}) that appear in the definition of the \textit{Maybe} type by:

\begin{verbatim}
> ghci> :kind Listify Symbol (Rep (Maybe Int))
> = '["Maybe","Nothing","Just"]
\end{verbatim}

With our generic framework now in place, we can finally revisit the \textit{HasField} and \textit{HasCtor} functions.

\begin{verbatim}

\textbf{type family HasCtor2} ctor f \textbf{where}
HasCtor2 ctor f = Foldl (||) 'False (Gmap (== ('MetaCons ctor)) f)
\end{verbatim}

We map the function \( (== \ '\text{MetaCons} \ ctor) \) over the structure. This implicitly selects only values of kind \textit{Meta}, and returns \textit{'True} for the constructors called \textit{ctor}. We then fold the result with the \((||)\) function, defaulting to \textit{False} in case the type had no constructors and \textit{Gmap} returned an empty list. This results in \textit{'True} if any of the constructors were called \textit{ctor}. \textit{Foldl} is simply the value-level \textit{foldl} function lifted to the type-level (higher-order, so it was not possible to write before):
Chapter 5. Unsaturated type families in practice

50

```
function 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:

```
function 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 (pronounced bind) passes the result of a stateful computation m a to a continuation a → 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:


Chapter 5. Unsaturated type families in practice

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 \( st \to st \) and applies it to the ambient state.

\[
\textbf{type family Modify } (f :: st \to st) :: \text{State } st () \text{ where } \\
\text{Modify } f = ' \text{State} \ (\text{Modify'} \ f)
\]

\[
\textbf{type family Modify'} (f :: st \to st) \ (s :: st) :: (st, ()) \text{ where } \\
\text{Modify'} \ f \ s = '(f \ s, '())
\]

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.

\[
\textbf{type Stateful } \\
= \text{Flip RunState } 0 \\
\quad \text{§ Modify } (\text{Add } 1) \\
\quad \gg \text{Modify } (\text{Add } 1) \\
\quad \gg \text{Modify } (\text{Add } 1) \\
\quad \gg \text{Modify } (\text{Add } 1) \\
\quad \gg \text{(Return } "\text{Works}" )
\]

Running the computation, we get

\[
\text{ghci} > :\text{kind Stateful} \\
\text{> = '(4,"Works")}
\]

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 re-purposed 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 \( \eta \)-rules.
Bibliography


