Higher-Order Type-Level Programming in Haskell

CSONGOR KISS, Imperial College London, United Kingdom
TONY FIELD, Imperial College London, United Kingdom
SUSAN EISENBACH, Imperial College London, United Kingdom
SIMON PEYTON JONES, Microsoft Research, United Kingdom

Type family applications in Haskell must be fully saturated. This means that all type-level functions have to be first-order, leading to code that is both messy and longwinded. In this paper we detail an extension to GHC that removes this restriction. We augment Haskell's existing type arrow, $\rightarrow$, with an *unmatchable* arrow, $\twoheadrightarrow$, that supports partial application of type families without compromising soundness. A soundness proof is provided. We show how the techniques described can lead to substantial code-size reduction (circa 80%) in the type-level logic of commonly-used type-level libraries whilst simultaneously improving code quality and readability.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Polymorphism**; *Data types and structures*; *Reusability*.

Additional Key Words and Phrases: Type-level programming, Type families, Higher-order functions

ACM Reference Format:
Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. 2019. Higher-Order Type-Level Programming in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 102 (August 2019), 26 pages. https://doi.org/10.1145/3341706

1 INTRODUCTION

Associated type families [Chakravarty et al. 2005] is one of *the* most widely-used of GHC's extensions to Haskell; in one study, type families was the third most-used extension (after overloaded strings and flexible instances) [Tondwalkar 2018]. In the example below, the type class *Db* classifies types, *a*, that can be converted into some primitive database type, *DbType a*, via a conversion function *toDb*. The type family *DbType* is a type-level function that takes the type *a* and returns the corresponding primitive database type that represents it. A type family instance is shown which states that a type *Username* will be represented by some database type *DbText*.

```
class Db a where
  type family DbType a
  toDb :: a → DbType a
```


Authors' addresses: Csongor Kiss, Department of Computing, Imperial College London, 180 Queen's Gate, London, SW7 1AZ, United Kingdom, cak14@imperial.ac.uk; Tony Field, Computing, Imperial College London, 180 Queen's Gate, London, SW7 1AZ, United Kingdom, ajf@imperial.ac.uk; Susan Eisenbach, Computing, Imperial College London, 180 Queen's Gate, London, SW7 1AZ, United Kingdom, susan@imperial.ac.uk; Simon Peyton Jones, Microsoft Research, 21 Station Road, Cambridge, CB1 2FB, United Kingdom, simonpj@microsoft.com.

2475-1421/2019/8-ART102
https://doi.org/10.1145/3341706

```
instance Db Username where
  type DbType Username = DbText
  toDb = (...)    -- The mapping function instance for Username (unspecified)
```

Despite their widespread use, type families come with a draconian restriction: they must be fully saturated. That is, a type family can only appear applied to all its arguments, never partially applied. We can have types like $T\ Maybe$, where $Maybe :: \star \to \star$ and $T :: (\star \to \star) \to \star$; and class constraints like $Monad\ IO$, where $IO :: \star \to \star$ and $Monad :: (\star \to \star) \to Constraint$. But the types $T\ DbType$ and $Monad\ Dbtype$ are not allowed, because $DbType$ is not saturated.

In the context of a higher-order functional language, this is most unfortunate. Higher-order functions are ubiquitous in term level programming, to support modularity and reduce duplication; for example, we define *sum* and *product* over lists by instantiating a generic *foldr* function.

Why can't we do the same for type functions? Because doing so would compromise both *type soundness* and efficient and predictable *type inference*. So we are between a rock and a hard place.

In this paper we show how to resolve this conflict by lifting the unsaturated type family restriction, so that we can, for the first time, perform higher-order programming at the type level in Haskell. Our extension brings the expressive power of Haskell's type language closer to the term language, and takes another important step towards bringing full-spectrum dependent types to Haskell [Weirich et al. 2017]. We make the following specific contributions:

- The type-level programming landscape has evolved greatly since the early days of Haskell 98, but however expressive, the type language remains first-order. In Section 2, we discuss the technical background behind the saturation restriction, and show how this restriction hinders the reusability of practical libraries.
- We describe an extension to Haskell's type system, called `UnsaturatedTypeFamilies` (Section 3), which lifts this restriction and unlocks sound abstraction over partial applications of type families. Originally suggested in [Eisenberg and Stolarek 2014], then developed in [Eisenberg 2016], we describe a new "non-matchable" arrow kind for type families ($\twoheadrightarrow$) that distinguishes type *constructors* (like *Maybe* or *Monad*) from type *families* (like *DbType*). Our contribution is to make a crucial further step: *matchability polymorphism* (Section 3.3). This allows type families to abstract uniformly over type constructors and other type families.
- To ensure that the resulting system is indeed sound, we present a statically-typed intermediate language, based closely on that already used in GHC, that supports full matchability polymorphism (Section 4). We prove type substitution and consistency lemmas and show that preservation and progress, and hence soundness, follow.
- Our system is no toy: we have implemented it in the Glasgow Haskell Compiler, GHC[1], as we describe in Section 5. We do not present formal results about type inference, but the changes to GHC's type inference engine are modest, and backward-compatible.
- We evaluate the new extension in Section 6, showing how it can describe universal notions of data structure traversal that can substantially reduce the volume of "boilerplate" code in type-level programs. When applied to the generic-lens library [Kiss et al. 2018], the type-level code is around 80% shorter than the original first-order equivalent; it is also higher level and easier to reason about.

We discuss related work in Section 7.

[1] https://gitlab.haskell.org/kcsongor/ghc/tree/icfp

2 TYPE FAMILIES AND TYPE-LEVEL PROGRAMMING IN HASKELL

2.1 Type Constructors and Type Families

In Haskell with type families there are three sorts of type constants:

- A *type constructor* is declared by a **data** or **newtype** declaration.
- A *type synonym* is declared by a top-level **type** declaration.
- A *type family* is declared by a **type** declaration inside a **class**, as in *DbType* above[2].

Here are some examples of type constructors and synonyms:

```
data Maybe a = Nothing | Just a    -- Type constructor
data Either a b = Left a | Right b -- Type constructor
type String = [Char]               -- Type synonym
```

The difference between *type constructors* and *type families* in types is similar to that between *data constructors* and *functions* in terms. The type (*Maybe Int*) is passive, and does not reduce, just like the term (*Just True*). But the type (*DbType Username*) can reduce to *DbText*, just as the function call *not True* can reduce to *False*.

Type family instances introduce new equality axioms and these are used in type inference. For example, the **type** instance declaration for *Db* above says that (*DbType Username*) and *DbText* are equal types. So, if $x :: Username$ and $f :: DbText \to IO\ ()$ then a call $f\ (toDb\ x)$ is well typed because $(toDb\ x)$ returns a *DbType UserName* and f expects an argument of type *DbText*; the types match as they are defined to be the same.

2.2 Injectivity and Generativity

Type constructors and families differ in two distinct ways: generativity and injectivity.

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

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

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

Type constructors, like *Maybe*, are both injective and generative; that is, they are *matchable* [Eisenberg 2016]. For example, suppose we know in some context that *Maybe Int* is equal to *Maybe a*, then we can conclude (by injectivity) that *a* must be equal to *Int*. The intuition here is that there is no other way to build the type *Maybe Int* other than to apply *Maybe* to *Int* (the type *Maybe Int* is canonical). In short *Maybe* is injective. Similarly, if we know in some context that *Maybe a* and $f\ b$ are equal, then we can conclude (by generativity) that f must be equal to *Maybe*.

What about type families? In contrast, they are *neither* injective *nor* generative[3]! For example, suppose that the database representation of *Email* in the example above is also *DbText*:

```
instance Db Email where
  type instance DbType Email = DbText
  toDb = (...)
```

DbType is clearly not injective, as we have defined *DbType Email* and *DbType Username* to be equal (they both reduce to *DbText*).

[2] In full Haskell a type family can also be declared with a *top-level* **type family** declaration; and such top-level declarations can be *open* or *closed* [GHC 2019]. Happily, the details of these variations are not important for this paper, and we stick only to *associated* type families, declared within a class.

[3] Haskell aficionados will know that the user can *declare* a type family to be injective [Stolarek et al. 2015]. But they cannot be generative, so from the perspective of this paper, declaring injectivity adds nothing.

2.3 Decomposing Type Applications

Injectivity and generativity have a profound influence on (a) type inference and (b) type soundness. We consider each in turn.

2.3.1 Inference. Consider the call $(f\ x)$, where $f :: \forall\ m\ a.\ Monad\ m \Rightarrow m\ a \rightarrow m\ a$, and $x :: F\ Int$ for some type family F. Is the call well-typed? We must instantiate f with suitable types t_m and t_a, and then we need to satisfy the "wanted" equality (see Section 5.1.1) $t_m\ t_a \sim F\ Int$, where $(\sim)$ means type equality. How can we do that? You might think that $t_m = F$ and $t_a = Int$ would work, and so it might. But suppose $F\ Int$ reduces to $Maybe\ Bool$; then $t_m = Maybe$ and $t_a = Bool$ would also work. Worse, if $F\ Int$ reduces to $Bool$ then the program is ill-typed.

So, during type inference, GHC never decomposes "wanted" equalities headed by a type family, like $t_m\ t_a \sim F\ Int$. But given a wanted equality like $t_m\ t_a \sim Maybe\ Int$ GHC does (and must) decompose it into two simpler wanted equalities $t_m \sim Maybe$ and $t_a \sim Int$, which are immediately soluble. Why must? Because if GHC does not decompose the equality it would end up with an unsolved equality and report a type error. To put it another way, decomposing matchable equalities is a key step in the standard Damas-Milner unification-based type inference algorithm.

2.3.2 Soundness. Is this function well typed, where F is a type family?

$$bad :: (F\ a \sim F\ b) \Rightarrow a \rightarrow b$$
$$bad\ x = x$$

To justify the definition $bad\ x = x$, we would need to prove the equality $(a \sim b)$. Can we prove it from $(F\ a \sim F\ b)$? That deduction would only be valid if F were injective. Type families are not in general injective, and it would be unsound to accept it. For example, if $F\ Char$ and $F\ Bool$ both reduce to the same thing then the call $bad\ \texttt{'x'} :: Bool$ would (erroneously) convert a $Char$ to a $Bool$ – by returning it unchanged!

To summarise, decomposing "wanted" equalities is sound, but leads to incomplete type inference; while decomposing "given" equalities is unsound. Accordingly, GHC only decomposes matchable equalities, i.e. those involving type constructors. (Reminder: type constructors were defined in Section 2.1.)

2.4 The Pain of Saturation

Now consider this variant of *bad*:

$$good :: \forall\ (f :: \star \rightarrow \star)\ a\ b.\ (f\ a \sim f\ b) \Rightarrow a \rightarrow b$$
$$good\ x = x$$

Can we decompose the given equality $(f\ a \sim f\ b)$ and hence justify the definition? GHC says "yes". But that is only sound if f is injective. So the question becomes: *how can we be sure that the type variable f will only be instantiated to an injective type?*

GHC's answer is simple: *type families must always appear saturated*, that is, applied to all their arguments, and hence *all well-formed types are injective and generative*. In effect this restricts us to *first-order* functional programming at the type level. A type-level function like *DbType* is not first class: it can only appear applied to its argument.

This is a painful restriction: at the term level, higher order functions (such as *map* and *foldr*) are one of the keys to modularity and re-use.

2.5 Use Case: *HList*s

To illustrate the pain of being stuck in a first-order world, we will look at heterogeneous lists [Kiselyov et al. 2004], which are widely used for implementing lists of objects of arbitrary type. Here is how a heterogeneous list type, *HList* say, can be defined as a GADT [Xi et al. 2003]:

```
data HList (xs :: [⋆]) where
  Nil  :: HList '[]
  (:>) :: a → HList as → HList (a ': as)
```

Using *HList* we can define a heterogeneous list of the attributes of a user, like this:

```
type User = '[Username, Password, Email, Date]

chris :: HList User
chris = Username "cc" :> Password "ahoy!" :> Email "cc@sm.com" :> Date 8 3 1492 :> Nil
```

The type (*HList User*) is indexed by *User*, a type-level list [Yorgey et al. 2012] of the types of the four attributes. Now suppose we have a *Db* instance of each of the attributes *Username*, *Password*, *Email*, and *Date* and we want to convert *chris* to its database representation by applying *toDb* to each field, like this, where T_1 and C_1 are place-holders for types we have yet to fill in:

```
dbChris :: HList T1
dbChris = mapToDb chris

mapToDb :: C1 ⇒ HList as → HList T1
mapToDb Nil       = Nil
mapToDb (a :> as) = toDb a :> mapToDb as
```

The code is obvious enough; what is tricky is the types. What can we write for the place-holders? Let us start with T_1. The function *mapToDb* applies *toDb* to each element of the list, so if the argument has type *HList* [*a*, *b*, *c*] then the result must have type *HList* [*DbType a*, *DbType b*, *DbType c*]. So *mapToDb* must have a type looking like this:

```
mapToDb :: C1 ⇒ HList as → HList (Map DbType as)
```

The easy bit is *Map*, the type-level version of *map*; we give its definition in Section 3.1. But the real problem is that *DbType* appears unsaturated which, as we have discussed, is simply not allowed in GHC.

We can make progress by writing a version of *Map* that is specialised to *DbType*, like this

```
type family MapDbType (xs :: [⋆]) :: [⋆] where
  MapDbType '[]       = '[]
  MapDbType (x ': xs) = DbType x ': MapDbType xs

dbChris   :: HList (MapDbType User)
mapToDb :: C1 ⇒ HList as → HList (MapDbType as)
```

Now *DbType* appears saturated. There is one more missing piece: what is C_1? The function *mapToDb* applies *toDb* to each element of the list, so it needs a (*Db t*) constraint for each type *t* in the argument list. Fortunately, GHC lets us compute constraints too [Bolingbroke 2011], like this:

```
type family All (c :: ⋆ → Constraint) (as :: [⋆]) :: Constraint where
  All c '[]       = ()
  All c (x ': xs) = (c x, All c xs)

mapToDb :: All Db as ⇒ HList as → HList (MapDbType as)
```

With these types, the code works. But there is a tremendous amount of boilerplate! All we are really doing is mapping a function down a list, both at the term level and the type level. Rather than hand-writing functions *mapToDb* and *MapDbType*, it would be far, far better to write something more like this (we'll complete the definition shortly):

$$
\begin{aligned}
&dbChris :: HList\ (Map\ DbType\ User)\\
&dbChris = hMap\ toDb\ chris\\
&\\
&hMap :: C_2 \Rightarrow T_2 \longrightarrow HList\ as \longrightarrow HList\ (Map\ f\ as)\\
&hMap\ f\ Nil \qquad = Nil\\
&hMap\ f\ (a :> as) = f\ a :> hMap\ f\ as
\end{aligned}
$$

for some C_2 and T_2, where *hMap* and *Map* are defined once and for all in libraries, rather than replicated by every client. Can we do that? Yes, we can.

– LocalWords: systemfc chak Db

3 THE SOLUTION: UNSATURATED TYPE FAMILIES

As we have seen, the trouble with unsaturated type families stems from the assumption that higher-order type variables stand for *generative* and *injective* type functions. Furthermore, type constructors, such as *Maybe*, have the same kind as type families, such as *DbType*, yet the latter must be avoided when decomposing equality constraints.

The solution is to distinguish type constructors from type families in the *kind system*, as first developed in [Eisenberg 2016] in the context of Dependent Haskell. That is, we distinguish the arrow of type constructors (matchable) from that of type families (unmatchable) and use two different symbols: $(\longrightarrow)$, i.e. Haskell's existing function arrow for the former, and $(\twoheadrightarrow)$ for the latter. Recall from Section 2.2 that *matchability* corresponds to functions that are both *generative* and *injective*. As an example, the kind of *Maybe* remains $\star \longrightarrow \star$, but *DbType* now has kind $\star \twoheadrightarrow \star$. Matchable applications can be decomposed, whereas unmatchable applications cannot.

Let us now revisit the function *good* from Section 2.4. In the equality constraint $f\ a \sim f\ b$, f has kind $\star \longrightarrow \star$, with a matchable arrow, so the constraint can be decomposed to give $a \sim b$, and that allows the right-hand side to typecheck. However, an attempt to instantiate f with *DbType* during type inference will now result in a kind error: f has kind $\star \longrightarrow \star$, but *DbType* has $\star \twoheadrightarrow \star$: their arrows don't match. On the other hand, if we modify *good* and attempt to abstract over *unmatchable* type functions instead:

$$
\begin{aligned}
&goodTry :: \forall\ (f :: \star \twoheadrightarrow \star)\ a\ b.\ (f\ a \sim f\ b) \Rightarrow a \longrightarrow b\\
&goodTry\ x = x
\end{aligned}
$$

we get a type error. This is because we cannot decompose unmatchable applications: $a \sim b$ is not derivable from $f\ a \sim f\ b$ because f here is defined to be unmatchable (its kind is $\star \twoheadrightarrow \star$). By separating matchable and unmatchable applications we have prevented the type system from constructing type equalities that break soundness.

The *good* function of Section 2.4 actually makes use only of injectivity, but not generativity. So why do we require the full the power of matchability when any injective function would do? Indeed, we could track injectivity and generativity separately by having dedicated arrows for both. This would enable abstraction over injective type families [Stolarek et al. 2015], but the practical applicability of such a scheme seems limited considering the additional complexity and notational burden it would incur.

We use the unmatchable function arrow $(\twoheadrightarrow)$ only in *kinds*, and not in *types*. For example, the type of a term like $id :: a \longrightarrow a$ still uses the matchable function arrow $(\longrightarrow)$, although morally *id* is

unmatchable. Luckily, this causes no problems, as matchability information is used only to guide decomposition of type equalities, based on their kinds.

3.1 HLists Revisited

Let us now return to the *HList* challenge in Section 2.4. The *Map* function used in the type of *dbChris* maps an *unmatchable* type function over a list of types:

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

By giving f the kind $a \twoheadrightarrow b$, we can write *Map DbType as* which is just what we needed for *mapToDb*. However, it prevents us from writing *Map Maybe as* because *Maybe* is injective: its kind is $\star \to \star$, not $\star \twoheadrightarrow \star$. This may seem unfortunate, but we can fix that too (Section 3.3).

We can now complete *hMap*'s type which had the form $C_2 \Rightarrow T_2 \to \mathit{HList}\ as \to \mathit{HList}\ (\mathit{Map}\ f\ as)$. Let us start with T_2. To make *hMap* completely general we have to abstract over the type family f and type class c that governs the function being mapped. For example, in *dbChris* above, f will be *DbType* and c will be *Db*. Each type that f is applied to must be an instance of c and that means T_2 must be a *rank-1* type: $(\forall\ a.\ c\ a \Rightarrow a \to f\ a)$, making the type of *hMap rank-2*.

What about C_2? Every element of the *HList* must be an instance of c, and we already have a type family, *All*, that computes this constraint. So, as with *Map*, C_2 is simply *All c as*. Thus, we arrive at:

$$\mathit{hMap} :: \mathit{All}\ c\ as \Rightarrow (\forall\ a.\ c\ a \Rightarrow a \to f\ a) \to \mathit{HList}\ as \to \mathit{HList}\ (\mathit{Map}\ f\ as)$$

In our implementation, GHC can infer the kinds of all the type variables, but it is instructive to see the same type signature, this time showing the bindings for c, f and *as*, and their kinds:

```
hMap :: ∀ (c :: ★ → Constraint) (f :: ★ ↠ ★) (as :: [★]).
          All c as ⇒ (∀ a. c a ⇒ a → f a) → HList as → HList (Map f as)
```

Here, we can see that f has an unmatchable function kind ($\star \twoheadrightarrow \star$), although this will be inferred anyway by virtue of the type of *Map*.

3.2 Visible Type Application

We need one additional fix to the definition of *dbChris* sketched in Section 2.4 above: we must pass c and f to *hMap* as explicit type parameters, thus:

```
dbChris :: HList (Map DbType User)
dbChris = hMap @Db @DbType toDb chris
```

The arguments "`@Db`" and "`@DbType`" explicitly instantiate c and f in *hMap*'s type, respectively. What on earth is going on here? Let us begin with a simpler example; suppose the *Db* class contained one more function, *size*:

```
class Db a where
  type family DbType a
  toDb :: a → DbType a
  size  :: DbType a → Int
```

and we want to typecheck a call (*size txt*) where *txt* :: *DbText*. The function *size* has type

$$\mathit{size} :: \forall\ a.\ \mathit{Db}\ a \Rightarrow \mathit{DbType}\ a \to \mathit{Int}$$

To typecheck the call (*size txt*) the type inference engine must determine what type should instantiate *a*; that choice will fix which *Db* dictionary is passed to *size*, which in turn determines

what (*size txt*) computes. But there may be many possible instantiations for *a*! For example, perhaps *DbType Username* and *DbType Email* are both *DbText*. We say that *size* has an *ambiguous type* because there is no unique way to infer a unique type instantiation from information about the argument and result types.

Functions with an ambiguous type can still be extremely useful, but to call such a function the programmer must supply the instantiation explicitly. In this example the programmer could write (*size* `@`*Username txt*) or (*size* `@`*Email txt*) to specify which instantiation they want. The "`@`*Email*" argument is called a *visible type argument*, and the language extension that supports visible type arguments is called *visible type application* [Eisenberg et al. 2016].

Using visible type application, the programmer is always *allowed* to supply such type arguments (e.g. *reverse* `@`*Bool* [*True*, *False*]), but if a function has an ambiguous type we *must* supply them. Returning to *hMap*, it certainly has an ambiguous type (because f appears only under a call to a type family *Map* and in an un-decomposable application $f\ a$), so we must supply f. There is a similar problem with *c*, which appears only in the constraint of the type. Hence the two type arguments in the call to *hMap* in *dbChris* above.

All of this applies equally to the recursive call in *hMap*'s own definition, so we must write:

```
hMap _ Nil       = Nil
hMap g (x :> xs) = g x :> hMap @c @f g xs
```

The alert reader will notice that, in both cases, we supplied only *two* of the three type arguments to *hMap*; that is, we explicitly instantiated *c* and *f*, but not *as*. It would be perfectly *legal* to supply a type argument for *as* as well, but it is not *necessary*, because it is not ambiguous. Moreover, it is slightly tiresome to specify: in *hMap*'s definition we would have to write

```
hMap g (x :> xs) = g x :> hMap @c @f @(Tail as) g xs
```

where *Tail* is a type family that takes the tail of a type-level list.

3.3 Matchability Polymorphism

Modifying the argument kind of *Map* allowed us to apply type families to the elements of the *HList*. However, what we gained on the swings, we lost on the roundabouts: *Map Maybe User* is a kind error due to the matchable arrow kind of *Maybe*. Ideally, we would like to be able to apply functions like *Map* to both type constructors and type families without having to duplicate *Map*'s definition.

At first you might think that we need subtyping, but instead we turn to polymorphism. Rather than having two separate arrows, we can use a single arrow $\to^m$ parameterised by its matchability. Its matchability m can be instantiated by M or U, for matchable and unmatchable respectively. The two arrows $\to$ and $\twoheadrightarrow$ now become synonyms for the two possible instantiations of $\to^m$:

```
type (→)  = →M
type (↠) = →U
```

M and U are ordinary data constructors

```
data Matchability = M | U
```

made available at the type level by GHC's `DataKinds` extension [Yorgey et al. 2012].

Now we can abstract over matchability to define a *matchability-polymorphic* version of *Map*:

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

The kind of *Map* thus becomes

$$Map :: \forall\ (m :: Matchability).\ (a\ \to^m\ b) \to\!\!\!\!\to [\,a\,] \to\!\!\!\!\to [\,b\,]$$

Similarly, *hMap*'s type can be generalised to accept both type families and type constructors:

$$\begin{aligned} hMap :: \forall\ \{\,m :: Matchability\,\}\ (c :: \star \to Constraint)\ (f :: \star\ \to^m \star)\ as. \\ \quad All\ as\ c \Rightarrow (\forall\ a.\ c\ a \Rightarrow a \to f\ a) \to HList\ as \to HList\ (Map\ f\ as) \end{aligned}$$

Note: the curly braces around $m :: Matchability$ means that it is an *inferred* argument; the visible-type-application mechanism does not apply to these inferred quantifiers. Otherwise, a call would have to look like *hmap* `@`*U* `@`*Db* `@`*DbType*..., with a tiresome extra explicit type argument `@`*U*[4].

It's not just type families that can abstract over matchabilities, but type constructors too. A popular technique in the Haskell folklore is to parameterise a data type by some functor, thereby fixing the general shape of the type while decorating the values in interesting ways. For example:

$$\textbf{data}\ T\ f = MkT\ (f\ Int)\ (f\ Bool)$$

By picking f to be *Maybe*, we get a version of T where each field is optional. By setting it to [ ], each field can store multiple values. By making T matchability-polymorphic and allowing type f to be instantiated with type families, we unlock whole new ways of doing abstraction:

$$\textbf{data}\ T\ (f :: \star\ \to^m \star) = MkT\ (f\ Int)\ (f\ Bool)$$

Here, T's kind becomes $T :: \forall\ m.\ (\star\ \to^m \star) \to \star$ so we can instantiate T either with a type constructor or a type family. For example, given the type family *Id*:

$$\begin{aligned} &\textbf{type family}\ Id\ a\ \textbf{where} \\ &\quad Id\ x = x \end{aligned}$$

we can have a version of T where the fields are simply *Int* and *Bool* (i.e. $T\ Id$), *Maybe Int* and *Maybe Bool* (i.e. $T\ Maybe$), or the database primitives *DbType Int* and *DbType Bool*, (i.e. $T\ DbType$).

4 FC_M: SYSTEM F_C WITH MATCHABILITY POLYMORPHISM

We now formalise our system as an extension of System F_C [Sulzmann et al. 2007], a small, explicitly-typed lambda calculus (*à la* Church) that is used as the intermediate language of GHC. Our system, FC_M, extends F_C with matchability polymorphism in types and kinds. Our main contribution is allowing partial application of type families, and showing that the desired progress and preservation properties of F_C are preserved by this change.

4.1 Syntax

The syntax of FC_M is shown in Figure 1 with the modifications to F_C highlighted. M stands for matchable, U stands for unmatchable, and m represents a matchability meta-variable (for matchability polymorphism).

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 $e\ \tau$. Type abstraction can now abstract over matchabilities. For example, the translation of *hMap* (Section 3.3) has the form $f = \Lambda(m :: Matchability).\ \Lambda(c :: \star \to Constraint).\ \Lambda(f :: \star\ \to^m \star).\ \Lambda(as :: [\star]).\ ....$

[4]This is tiresome because the instantiation of m can always be inferred from the instantiation of f. "Inferred" quantification will soon be allowed by GHC, as described in a current proposal https://github.com/ghc-proposals/ghc-proposals/pull/99.

Metavariables:

x term	a, b type	c coercion
C axiom	F_n n-ary type family	m matchability

e, u	$::=$	expressions
	$\mid x$	variables
	$\mid \lambda x{:}\tau.\, e \mid e_1\ e_2$	abstraction/application
	$\mid \Lambda a{:}\kappa.\, e \mid e\ \tau$	type abstraction/application
	$\mid \lambda c{:}\phi.\, e \mid e\ \gamma$	coercion abstraction/application
	$\mid e \triangleright \gamma$	cast
κ	$::=$	kinds
	$\mid \star$	base kind
	$\mid$ MATCHABILITY	matchability kind
	$\mid \kappa_1 \rightarrow^{\nu} \kappa_2$	arrow kind
	$\mid \forall m.\, \kappa$	matchability polymorphism
τ, σ, ν	$::=$	types
	$\mid a$	variables
	$\mid \tau \rightarrow \sigma$	abstraction
	$\mid \phi \Rightarrow \sigma$	coercion abstraction
	$\mid \tau_1\ \tau_2$	type application
	$\mid \forall a{:}\kappa.\, \tau$	type polymorphism
	$\mid H$	type constants
H	$::=$	type constants
	$\mid T$	type constructors
	$\mid F$	type families
	$\mid M$	Matchable
	$\mid U$	Unmatchable
ϕ	$::= \tau \sim \sigma$	propositions (coercion kinds)
γ	$::=$	coercions
	$\mid c$	variables
	$\mid$ **refl** $\tau \mid$ **sym** $\gamma \mid$ **trans** $\gamma_1\ \gamma_2$	equivalence
	$\mid \gamma_1 \rightarrow \gamma_2$	arrow type congruence
	$\mid \phi \Rightarrow \gamma$	coercion arrow type congruence
	$\mid \gamma_1\ \gamma_2$	type application congruence (in types)
	$\mid \gamma\ \nu$	type application congruence (in kinds)
	$\mid \forall a{:}\kappa.\, \gamma$	polytype congruence
	$\mid C(\overline{m}, \overline{\gamma})$	axiom application
	$\mid$ **left** $\gamma \mid$ **right** γ	decomposition
	$\mid \gamma\ @\ \tau$	type instantiation

Fig. 1. Syntax of FC_M with matchability extensions highlighted.

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

$$\frac{\vdash \Gamma \qquad x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau}\text{E\_Var} \qquad \frac{\Gamma, x{:}\tau \vdash e{:}\sigma}{\Gamma \vdash \lambda x{:}\tau.e{:}\tau \to \sigma}\text{E\_Abs} \qquad \frac{\Gamma \vdash e_1{:}\tau \to \sigma \qquad \Gamma \vdash e_2{:}\tau}{\Gamma \vdash e_1\ e_2{:}\sigma}\text{E\_App}$$

$$\frac{\Gamma, a{:}\kappa \vdash e{:}\tau}{\Gamma \vdash \Lambda a{:}\kappa.e{:}\forall a{:}\kappa.\tau}\text{E\_TAbs} \qquad \frac{\Gamma \vdash e{:}\forall a{:}\kappa.\sigma \qquad \Gamma \vdash_{\text{ty}} \tau{:}\kappa}{\Gamma \vdash e\ \tau{:}\sigma[\tau/a]}\text{E\_TApp}$$

$$\frac{\Gamma, c : \phi \vdash e : \tau}{\Gamma \vdash \lambda c : \phi.e : \phi \Rightarrow \tau}\text{E\_CAbs} \qquad \frac{\Gamma \vdash e : \phi \Rightarrow \tau \qquad \Gamma \vdash_{\text{co}} \gamma : \phi}{\Gamma \vdash e\ \gamma : \tau}\text{E\_CApp}$$

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

Fig. 2. Expression typing

4.2 Typing Rules

The expression typing rules for FC_M are shown in Figure 2.

Kinds include the base kind $\star$, the kind of matchabilities, arrow kinds parameterised by their matchabilities, and matchability quantification. Types with matchability polymorphic arrow kinds can be instantiated by the $\tau\ \nu$ application form. The corresponding typing rules are shown in Figure 3 (Ty_Inst).

The original System F_C syntactically distinguished type family applications (which are fully saturated) from other type applications. We remove this distinction, which means that both type family and type constructor applications are represented by the $\tau_1\ \tau_2$ form. The corresponding typing rule in Figure 3 is Ty_App, which is now polymorphic in the matchability of τ_1's kind.

Figure 4 shows the valid typing contexts. Note that a type constructor's kinds can quantify over matchability variables; for example in Section 3.3 we saw $T :: \forall\ m.\ (\star \to^m \star) \to \star$.

To reduce clutter our system supports *matchability polymorphism*, but not *kind polymorphism*; for example, it does not support $T :: \forall\ k.\ k \to \star$. There is no difficulty in combining the two, however, and our implementation does so.

4.3 Coercions

One of the main innovations of System F_C is the use of *coercions*, or type equalities. A coercion $\phi = \tau \sim \sigma$ represents (homogeneous) type equality between τ and σ. Coercions can be abstracted over with $\lambda c{:}\phi.e$ and applied as $e\ \gamma$.

Non-syntactic equalities, such as those introduced by type family equations, pose a challenge in compilation, as they make it difficult to do sanity checks on the intermediate representation. Coercions solve this problem by reifying the type equality derivations and encoding them into the terms themselves. What this means is that the only way to convert an expression e of type τ_1 into type τ_2 is by providing an explicit witness (a *coercion*) γ of the type equality $\tau_1 \sim \tau_2$ and explicitly *casting* e by γ, *viz.* $e \triangleright \gamma$. The corresponding typing rule is E_Cast in Figure 2.

Figure 5 displays the formation rules for coercions. They are a syntactic reification of the equivalence relationship (with corresponding reflexivity (Co_Refl), symmetry (Co_Sym), and transitivity (Co_Trans) rules) with congruence. This way, type checking in F_C is *syntactic*, as

$$\boxed{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}$$

$$\frac{a : \kappa \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} a : \kappa}\textsc{Ty\_Var} \qquad \frac{T : \kappa \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} T : \kappa}\textsc{Ty\_Data} \qquad \frac{F_n : \kappa \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} F_n : \kappa}\textsc{Ty\_Fun}$$

$$\frac{}{\Gamma \vdash_{\mathsf{ty}} M : \textsc{Matchability}}\textsc{Ty\_Matchable} \qquad \frac{}{\Gamma \vdash_{\mathsf{ty}} U : \textsc{Matchability}}\textsc{Ty\_Unmatchable}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \to^{\nu} \kappa_2 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau_1\ \tau_2 : \kappa_2}\textsc{Ty\_App} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \tau : \forall m{:}\kappa_1.\,\kappa \qquad \Gamma \vdash_{\mathsf{ty}} \nu : \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau\ \nu : \kappa[\nu/m]}\textsc{Ty\_Inst}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \sigma, \tau : \star}{\Gamma \vdash_{\mathsf{ty}} \sigma \to \tau : \star}\textsc{Ty\_Arrow} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \tau : \star \qquad \Gamma \vdash_{\mathsf{ty}} \sigma_1, \sigma_2 : \kappa}{\Gamma \vdash_{\mathsf{ty}} \sigma_1 \sim \sigma_2 \Rightarrow \tau : \star}\textsc{Ty\_CArrow}$$

$$\frac{\Gamma, a{:}\kappa \vdash_{\mathsf{ty}} \tau : \star \qquad \Gamma \vdash_{\mathsf{k}} \kappa}{\Gamma \vdash_{\mathsf{ty}} \forall a{:}\kappa.\,\tau : \star}\textsc{Ty\_All}$$

$$\boxed{\Gamma \vdash_{\mathsf{k}} \kappa}$$

$$\frac{}{\Gamma \vdash_{\mathsf{k}} \star}\textsc{K\_Star} \qquad \frac{}{\Gamma \vdash_{\mathsf{k}} \textsc{Matchability}}\textsc{K\_Matchability}$$

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa_1 \qquad \Gamma \vdash_{\mathsf{k}} \kappa_2 \qquad \Gamma \vdash_{\mathsf{ty}} \nu : \textsc{Matchability}}{\Gamma \vdash_{\mathsf{k}} \kappa_1 \to^{\nu} \kappa_2}\textsc{K\_Arrow} \qquad \frac{\Gamma, m{:}\kappa_1 \vdash_{\mathsf{k}} \kappa \qquad \Gamma \vdash_{\mathsf{k}} \kappa_1}{\Gamma \vdash_{\mathsf{k}} \forall m{:}\kappa_1.\,\kappa}\textsc{K\_All}$$

Fig. 3. Type kinding

$$\Gamma \quad ::= \varnothing \mid \Gamma, bnd$$

bnd	$::=$	binders
	$\mid x : \tau$	term variable
	$\mid a : \kappa$	type variable
	$\mid c : \phi$	coercion variable
	$\mid T : \forall \overline{m}.\overline{\kappa} \to^{M} \star$	data type
	$\mid F_n : \forall \overline{m}.\ \overline{\kappa}^n \to^{U} \kappa$	n-ary type family
	$\mid C(\overline{m}, \overline{a : \kappa}) : \phi$	axioms

Fig. 4. Contexts

all the derivations are encoded in the terms via casts. For example, in order to use a coercion $\gamma : \mathit{Bool} \sim a$ to prove that $e : a$ resolves to $e : \mathit{Bool}$, we explicitly cast e via $e \triangleright \mathbf{sym}\ \gamma$.

Coercions can be *decomposed*, which is crucial for type inference. The **left** and **right** coercions in Figure 5 split apart an equality between application forms into their constituent parts, as shown in Co_Left and Co_Right respectively. Since both type family applications and type constructor applications are represented by the $\tau_1\ \tau_2$ form, we augment these rules by the additional premise that the function must have a matchable kind. This is in order to ensure consistency (Section 4.5.2).

What happens if we omit the highlighted premise? Presumably we can derive a bogus equality. To see how, consider the translation of the *goodTry* function from Section 3:

$$\begin{aligned} \mathit{goodTry} = \Lambda(f : \star \twoheadrightarrow \star)\ (a : \star)\ (b : \star).\quad & \text{-- NB: (right co) is ill-typed} \\ \lambda(\mathit{co} : f\ a \sim f\ b).\ \lambda(x : a).\ & (x \triangleright \mathbf{right}\ \mathit{co}) \end{aligned}$$

The return type of *goodTry* is b, but it just returns its argument, which is of type a. Thus, we need to find evidence that a can be cast into b. Decomposing the assumed *co* coercion using **right**, we get a coercion of type $a \sim b$. Now, assuming the top-level environment contains the following axioms:

$$\begin{aligned} &\mathbf{axiom}\ \mathit{db1} :: \mathit{DbType\ Username} \sim \mathit{DbText} \\ &\mathbf{axiom}\ \mathit{db2} :: \mathit{DbType\ Email} \sim \mathit{DbText} \end{aligned}$$

then we can compose these coercions using *transitivity*, and *symmetry* of *db2*:

$$\mathbf{trans}\ \mathit{db1}\ (\mathbf{sym}\ \mathit{db2}) :: \mathit{DbType\ Username} \sim \mathit{DbType\ Email}$$

The problem happens when we now instantiate the arguments to *goodTry* by $f := \mathit{DbType}$, $a := \mathit{Username}$, $b := \mathit{Email}$, $\mathit{co} := \mathbf{trans}\ \mathit{db1}\ (\mathbf{sym}\ \mathit{db2})$. This will produce the coercion

$$\mathbf{right}\ (\mathbf{trans}\ \mathit{db1}\ (\mathbf{sym}\ \mathit{db2})) : \mathit{Username} \sim \mathit{Email}$$

which is clearly inconsistent. By enforcing the matchable arrow kind the coercion **right** *co* in *goodTry* becomes ill-typed, because *co* relates functions of unmatchable kinds and so cannot be decomposed by **right**.

Finally, the coercion language includes *axioms* (such as *db1* and *db2* above), of the form $C(\overline{m}, \overline{\tau : \kappa}) : \phi$. Such axioms are introduced by type family equations. Axiom applications are written in a first-order way to emphasise that they are always to be saturated. This is not a limitation, however. Every type family equation introduces a new axiom, and the arguments of a type family application determine which axiom to use. This means that we can only pick the matching axiom once the type family is fully saturated. This is not surprising, as we wouldn't expect a partially applied function to reduce.

4.4 Operational Semantics

The operational semantics of FC_M is unchanged from that of System F_C, with the exception of a standard beta rule for matchability abstraction: We therefore omit further details, but we show the necessary substitution lemmas which are needed for preservation.

4.5 Metatheory

We now show that our system enjoys the usual metatheoretic properties such as progress and preservation. The main difference from System F_C is that type variables can now be instantiated with unsaturated applications of type families, and we need to ensure that type safety is not violated by lifting this restriction. Since coercion axioms give rise to a non-trivial equational theory, we must ensure that the coercion relation is consistent with respect to the top-level axioms.

$$\boxed{\Gamma \vdash_{\text{co}} \gamma : \phi}$$

$$\frac{\Gamma \vdash_{\text{ty}} \tau : \kappa}{\Gamma \vdash_{\text{co}} \textbf{refl}\ \tau : \tau \sim \tau}\text{Co\_Refl} \qquad \frac{\Gamma \vdash_{\text{co}} \gamma : \sigma \sim \tau}{\Gamma \vdash_{\text{co}} \textbf{sym}\ \gamma : \tau \sim \sigma}\text{Co\_Sym}$$

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

$$\frac{\Gamma \vdash_{\text{co}} \gamma_1 : \tau_1 \sim \sigma_1 \qquad \Gamma \vdash_{\text{co}} \gamma_2 : \tau_2 \sim \sigma_2}{\Gamma \vdash_{\text{co}} \gamma_1 \to \gamma_2 : (\tau_1 \to \sigma_1) \sim (\tau_2 \to \sigma_2)}\text{Co\_Arrow}$$

$$\frac{\Gamma \vdash_{\text{co}} \gamma : \tau \sim \sigma}{\Gamma \vdash_{\text{co}} \phi \Rightarrow \gamma : (\phi \Rightarrow \tau) \sim (\phi \Rightarrow \sigma)}\text{Co\_CArrow} \qquad \frac{\Gamma \vdash_{\text{ty}} \sigma_1, \sigma_2 : \kappa_1 \to^{\nu} \kappa_2 \qquad \Gamma \vdash_{\text{ty}} \tau_1, \tau_2 : \kappa_1 \qquad \Gamma \vdash_{\text{co}} \gamma_1 : \sigma_1 \sim \sigma_2 \qquad \Gamma \vdash_{\text{co}} \gamma_2 : \tau_1 \sim \tau_2}{\Gamma \vdash_{\text{co}} \gamma_1\ \gamma_2 : \sigma_1\ \tau_1 \sim \sigma_2\ \tau_2}\text{Co\_App}$$

$$\frac{\Gamma \vdash_{\text{ty}} \tau_1, \tau_2 : \forall m : \kappa_1.\ \kappa \qquad \Gamma \vdash_{\text{co}} \gamma : \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\text{ty}} \nu : \kappa_1}{\Gamma \vdash_{\text{co}} \gamma\ \nu : \tau_1\ \nu \sim \tau_2\ \nu}\text{Co\_App\_K}$$

$$\frac{\Gamma, a : \kappa \vdash_{\text{ty}} \tau, \sigma : \star \qquad \Gamma, a : \kappa \vdash_{\text{co}} \gamma : \tau \sim \sigma}{\Gamma \vdash_{\text{co}} \forall a : \kappa.\ \gamma : \forall a : \kappa.\ \tau \sim \forall a : \kappa.\ \sigma}\text{Co\_Abs} \qquad \frac{\Gamma \vdash_{\text{ty}} \sigma_1, \sigma_2 : \kappa_1 \to^{M} \kappa_2 \qquad \Gamma \vdash_{\text{co}} \gamma : \sigma_1\ \tau_1 \sim \sigma_2\ \tau_2}{\Gamma \vdash_{\text{co}} \textbf{left}\ \gamma : \sigma_1 \sim \sigma_2}\text{Co\_Left}$$

$$\frac{\Gamma \vdash_{\text{ty}} \sigma_1, \sigma_2 : \kappa_1 \to^{M} \kappa_2 \qquad \Gamma \vdash_{\text{co}} \gamma : \sigma_1\ \tau_1 \sim \sigma_2\ \tau_2}{\Gamma \vdash_{\text{co}} \textbf{right}\ \gamma : \tau_1 \sim \tau_2}\text{Co\_Right} \qquad \frac{\Gamma \vdash_{\text{co}} \gamma : \forall a : \kappa.\tau_1 \sim \forall a : \kappa.\sigma_1 \qquad \Gamma \vdash_{\text{co}} \tau : \kappa}{\Gamma \vdash_{\text{co}} \gamma\ @\ \tau : \tau_1[\tau/a] \sim \sigma_1[\tau/a]}\text{Co\_Inst}$$

$$\frac{\begin{array}{c} C(\overline{m}, \overline{a : \kappa}) : \tau_1 \sim \tau_2 \in \Gamma \\ \text{for each } \gamma_i \in \overline{\gamma}, \\ \Gamma \vdash_{\text{co}} \gamma_i : \sigma_{1i} \sim \sigma_{2i} \\ \Gamma \vdash_{\text{ty}} \sigma_{1i}, \sigma_{2i} : \kappa_i[\overline{\nu/m}] \end{array}}{\Gamma \vdash_{\text{co}} C(\overline{\nu}, \overline{\gamma}) : \tau_1[\overline{\nu/m}][\overline{\sigma_1/a}] \sim \tau_2[\overline{\nu/m}][\overline{\sigma_2/a}]}\text{Co\_Axiom}$$

Fig. 5. Formation rules for coercions

We discuss sufficient requirements for top-level contexts to be *consistent*, and show how the typing judgments can be guarded against deriving inconsistent conclusions – both are key for the progress theorem. Our proofs extend previous work on matchability [Eisenberg 2016] with the additional treatment of matchability polymorphism.

4.5.1 Preservation. Our extension of the operational semantics is uninteresting, so the preservation proof is standard [Sulzmann et al. 2007]. The steps in the operational semantics preserve the types, so the only thing we need to ensure is that, in the case of β-reductions, the substitutions are type preserving.

The following lemmas state that coercion derivations are preserved by type, matchability, and coercion substitution, and they can be proved by induction on the height of the derivations.

Lemma (Matchability substitution in kinds). *If* $\Gamma_1, m, \Gamma_2 \vdash_k \kappa$ *and* $\Gamma_1 \vdash_m \nu$ *then* $\Gamma_1, \Gamma_2[\nu/m] \vdash_k \kappa[\nu/m]$

Lemma (Type substitution in coercions). *If* $\Gamma_1, (a:\kappa), \Gamma_2 \vdash_{co} \gamma_1 : \tau_1 \sim \tau_2$ *and* $\Gamma_1 \vdash_{ty} \tau : \kappa$ *then* $\Gamma_1, \Gamma_2[\tau/a] \vdash_{co} \gamma_1[\tau/a] : \tau_1[\tau/a] \sim \tau_2[\tau/a]$

Lemma (Coercion substitution in coercions). *If* $\Gamma_1, (c:\phi_1), \Gamma_2 \vdash_{co} \gamma_1 : \phi_2$ *and* $\Gamma_1 \vdash_{co} \gamma_2 : \phi_1$ *then* $\Gamma_1, \Gamma_2[\gamma_2/c] \vdash_{co} \gamma_1[\gamma_2/c] : \phi_2$

Similar substitution lemmas can be proved for terms. Given a top-level environment Σ, the preservation theorem follows:

Theorem (Preservation). *If* $\Sigma \vdash_{tm} e_1 : \tau$ *and* $e_1 \longrightarrow e_2$ *then* $\Sigma \vdash_{tm} e_2 : \tau$

The top-level environment Σ contains only type family signatures, type constructor signatures, and coercion axioms.

4.5.2 Progress. The progress proof also follows previous work, but it requires that the top-level environment is *consistent*. That is, all derivable coercions preserve the head forms of types. In other words, it is not possible to derive bogus equalities like $Char \sim Bool$. Ensuring that this assumption holds is our primary concern here.

Definition (Value type). *A type* τ *is a value type in an environment* Γ *iff*

- $\Gamma \vdash_{ty} \tau : \star$
- τ *is of the form* $T\ \overline{m}\ \overline{\sigma_1}$ *or* $(\sigma_1 \to \sigma_2)$ *or* $(\phi_1 \Rightarrow \sigma_1)$ *or* $(\forall a:\kappa.\ \sigma_1)$ *or* $(\forall m.\ \sigma_1)$

Notably, type family applications are *not* value types.

Definition (Consistency). *A context* Γ *is consistent iff*

- *If* $\Gamma \vdash_{co} \gamma : T\ \overline{m}\ \overline{\sigma_1} \sim \tau$ *and* τ *is a value type, then* $\tau = T\ \overline{m}\ \overline{\sigma_2}$
- *If* $\Gamma \vdash_{co} \gamma : \sigma_1 \to \sigma_2 \sim \tau$ *and* τ *is a value type, then* $\tau = \sigma_3 \to \sigma_4$
- *If* $\Gamma \vdash_{co} \gamma : \phi_1 \Rightarrow \sigma_1 \sim \tau$ *and* τ *is a value type, then* $\tau = \phi_2 \Rightarrow \sigma_2$
- *If* $\Gamma \vdash_{co} \gamma : (\forall a:\kappa.\ \sigma_1) \sim \tau$ *and* τ *is a value type, then* $\tau = (\forall a:\kappa.\ \sigma_2)$
- *If* $\Gamma \vdash_{co} \gamma : (\forall m.\ \sigma_1) \sim \tau$ *and* τ *is a value type, then* $\tau = (\forall m.\ \sigma_2)$

That is, we require that all coercion derivations preserve the outermost constructors. Consistency might be imperiled by two factors: bogus axioms in the top-level environment (such as $Char \sim Bool$), and inconsistent coercion derivations. The latter might happen if we try to decompose a non-injective function application.

To summarise, consistency is a property of not just the top-level environment, but the coercion judgements too. We consider each in turn.

4.5.3 Consistency of Top-Level Environment. Type family axioms introduce arbitrary equalities. To ensure they are consistent, we need to place restrictions on the equations. We require the following conditions:

(1) All axioms are of the form $c : \forall \overline{m}.\ F_n\ \overline{m}\ \overline{\tau : \kappa}^n \sim \sigma$. The type patterns $\overline{\tau : \kappa}^n$ must mention no type families, and all type variables must be distinct. Furthermore, all variables $\overline{m}$ must appear free in at least one of the kinds $\overline{\kappa}$ of the patterns. Lastly, all type applications in patterns must be headed by matchable type functions.
(2) There is no overlap between axioms: given $F_n\ \overline{m}\ \overline{\tau}^n$, there exists at most one axiom C such that $C(\overline{m}, \overline{\tau}^n) : F_n\ \overline{m}\ \overline{\tau}^n \sim \sigma$.

The restrictions on patterns is standard. An unusual feature of type families is that they can match on unknown type constructor applications. For example:

```
type family Match f
type instance Match (f a) = a
```

The restriction that type applications must be headed by matchable functions means that f cannot be a type family.

As in previous work [Weirich et al. 2011], type families can be interpreted as a parallel reduction relation, which, when restricted in the way described above, can be shown to be locally confluent. Then, by assuming termination of the rewrite system, we appeal to *Newman's lemma* to show confluence of the rewrite system.

4.5.4 Consistency of Coercion Judgements. A crucial difference from System F_C is that type variables can be instantiated to unsaturated type families. The Co_Right rule in Figure 5 ensures that functions of unmatchable kinds cannot be decomposed by **right**.

Lemma (Coercion judgement consistency). *If the axioms in Σ define a confluent rewriting system, then Σ is consistent.*

The proof requires showing that the coercion judgements preserve head forms. This can be done by induction on the height of the derivations. The Co_Refl and Co_Sym rules are straightforward. In Co_Trans, we appeal to the induction hypotheses. Co_Var is vacuously true, because we're in the top-level environment where no coercion variables are bound. The congruence rules Co_App, Co_MApp, Co_Abs, and Co_MAbs are similarly straightforward. Co_Left and Co_Right require that the constructors are matchable, thus they have the necessary generativity and injectivity properties. For Co_Inst and Co_Inst_M, we appeal to the respective substitution lemmas.

Now, assuming that the top-level environment Σ is consistent, the progress theorem follows:

Theorem (Progress). *If $\Sigma \vdash_{tm} e_1 : \tau$, then e_1 is either a value, or there is some e_2, such that $e_1 \longrightarrow e_2$.*

5 PRACTICALITIES

We have discussed matchability polymorphism as an extension to GHC's core calculus, and we now turn to some of the practical aspects of integrating matchabilities into the source language.

We have a fork of GHC that implements a new language extension, `UnsaturatedTypeFamilies`, which supports all the features described in this paper, including their interaction with GADTs, data families, pattern synonyms, etc which we have not described at all. Our language design is backward-compatible, so that all existing Haskell programs continue to work, even when the `UnsaturatedTypeFamilies` extension is enabled. Moreover, our prototype is sufficiently robust to bootstrap GHC itself and compile a large suite of libraries. All the examples in this paper are accepted by our prototype.

In this section we review some highlights of our implementation experience.

5.1 Type Inference

Unlike FC_M, source Haskell is an *implicitly* typed language, which means that (most) type annotations are optional, as they can be inferred by a compiler. Type inference is the process of elaborating Haskell code into the explicitly typed FC_M.

GHC already has a powerful type inference engine, and it turned out that the extensions needed for this paper fitted neatly into the existing framework. In particular:

- Matchabilty affects the decomposition of equalities, which takes place in GHC's constraint solver. Restricting decomposition to matchable arrows was mostly a matter of adding a guard to that code.
- Potentially harder is matchability polymorphism. Happily, as well as traditional *type* polymorphism, GHC also accommodates *kind* polymorphism [Yorgey et al. 2012] and *levity* polymorphism [Eisenberg and Peyton Jones 2017]. So a fourth flavour of polymorphism came almost for free[5]. The same, existing, constraint solver uniformly solves type, kind, levity, and now matchability constraints.

5.1.1 Constraint-Solving. GHC uses a powerful *constraint-based* type inference algorithm called OutsideIn(X) [Vytiniotis et al. 2011]. The algorithm is conceptually simple: it generates constraints from the source language, then solves these constraints [Simonet and Pottier 2007]. In our case, the constraints are type equalities, and the solutions are encoded into coercions witnessing the equalities (Section 4.3). Constraints come in two flavours: *Given*, and *Wanted*.[6] For example, consider this definition

$$\begin{aligned} &bar :: f \sim Id \Rightarrow f\ Bool \\ &bar = False \end{aligned}$$

where Id is the identity type family (Section 3.3). The assumption $f \sim Id$ is a *Given* constraint, which can be used to solve the *Wanted* constraint $Bool \sim f\ Bool$, which arises from matching the type of *bar*'s body with *bar*'s declared type. The first step is inferring the kind of f. Since it is used in an application $f\ Bool$, it must have an arrow kind. For its matchability, the constraint solver invents a fresh unification variable $\alpha :: Matchability$, thus $f :: \star \to^{\alpha} \star$. Unifying the kind of f with that of Id produces the substitution $\alpha := U$. Therefore in this example, we can infer that f's kind is $\star \to\!\!\!\!\to \star$.

All this fits in beautifully with GHC's existing mechanism.

5.1.2 Generalising Over Matchability. Consider this type signature:

$$\begin{aligned} &\textbf{class}\ Functor\ f\ \textbf{where} \\ &\quad fmap :: (a \to b) \to f\ a \to f\ b \end{aligned}$$

What kind should we infer for f? The most general answer is this:

$$\begin{aligned} &\textbf{class}\ Functor\ (f :: \star \to^{m} \star)\ \textbf{where} \\ &\quad fmap :: (a \to b) \to f\ a \to f\ b \end{aligned}$$

That is, *Functor* becomes matchability-polymorphic, with kind

$$Functor :: \forall\ (m :: Matchability).\ (\star \to^{m} \star) \to Constraint$$

That *might* be what the programmer intended, but it is a perplexing kind to show to the programmer. Moreover, if matchability polymorphism becomes pervasive, more types will become ambiguous,

[5] As always, "for free" simply means "already paid for".

[6] GHC's implementation also has a notion of "derived" constraints, but we do not discuss them here.

so silent matchability polymorphism is not necessarily a good thing, even if the programmer never saw it. For example, consider the following function:

$$silly :: f\ Int \to f\ Int$$
$$silly = id$$

The most general kind for f is the matchability-polymorphic $\star \to^m \star$. Here, f is ambiguous: what should it be in the expression *silly* (*Just* 10)? We have a Wanted equality $f\ Int \sim Maybe\ Int$, but it cannot be decomposed, because f is not known to be matchable, so we are stuck.

So our choice is this: when automatically generalising the kind of a type, or the type of a term, we never generalise over a matchability variable. Instead of generalising, we "default" any unconstrained matchability variables to M, which is the choice for *all* types in legacy Haskell. So *Functor* will get the more familiar kind

$$Functor :: (\star \to \star) \to Constraint$$

If the programmer wants matchability polymorphism, they must declare it – and GHC has perfectly adequate mechanisms to allow them to do so.

Matchability defaulting also allows the constraint solver to make progress when it gets stuck on a decomposition problem due to polymorphism. Suppose that we really want our function *silly* to have a polymorphic type:

$$silly :: \forall\ m\ (f :: \star \to^m \star).\ f\ Int \to f\ Int$$
$$silly = id$$

As before, *silly* (*Just* 10) gets stuck when trying to solve $f\ Int \sim Maybe\ Int$. Since there are no constraints in the context that would determine the matchability of f, it is *unconstrained*. So matchability defaulting can make progress: by setting $f :: \star \to \star$, the equality can now be decomposed and f instantiated to *Maybe*.

This design choice replicates a similar choice in the realm of *levity polymorphism* [Eisenberg and Peyton Jones 2017], where it has proven to be robust.

5.1.3 Occurs Checking. In order to avoid infinite cycles, GHC employs a syntactic occurrence check to rule out erroneous type equalities during unification. For example, the equation

$$\alpha \sim Maybe\ \alpha$$

is insoluble, because the metavariable α *occurs* on the right hand side, and setting $\alpha := Maybe\ \alpha$ would lead to an infinite substitution. The grounds for rejection in this case is that *Maybe* is a *generative* type constructor, so the equation cannot possibly hold.

However, if the variable α is applied to a type family on the right hand side, then there might exist a solution.

$$\alpha \sim Id\ \alpha$$

Here, even though α occurs on the right hand side, it only does so as an argument to the *Id* type family, and so this equation is indeed valid.

The final case is when the variable is applied to another variable

$$\alpha \sim \beta\ \alpha$$

Today, GHC rejects this equation because of the assumption that all type families appear saturated, so there can be no other equation whose solution is to set β to a type family like *Id*. Of course, this assumption no longer holds when `UnsaturatedTypeFamilies` is enabled, so we modified the occurs checker to take into account the matchability of β. The equation is definitely insoluble just when β has a matchable arrow, but otherwise a solution might exist.

5.2 Interaction with Kind Equalities

In GHC's type system, kinds (like $\star \to \star$) are types, constraints (like *Eq a*) are types, levities are types, and (now) matchabilities are types. For example the type *Int* and matchability *M* are both types; they are distinguished only by their kinds ($\star$ and *Matchability* respectively). This means that the entire apparatus of type inference, classes, type families and so on all works equally well on matchabilities.

For example, it is possible to write a type family that returns the matchability of its argument:

```
type family MatchabilityOf (f :: a →^m b) :: Matchability where
  MatchabilityOf (_ :: _ →^m _) = m
```

Conversely, it is also possible to compute matchabilities based on type information. As a contrived example, *F b* returns a matchable type just when *b* is *True*, and an unmatchable one otherwise:

```
type family F (b :: Bool) :: ★ →^(G b) ★ where
  F 'True  = Maybe
  F 'False = DbType
```

where the matchability is computed by another type family, *G*:

```
type family G (b :: Bool) :: Matchability where
  G 'True  = 'M
  G 'False = 'U
```

6 UNSATURATED TYPE FAMILIES IN PRACTICE: A CASE STUDY

One of the original inspirations for this paper was the generic-lens library [Kiss et al. 2018] code, which is designed to decrease boilerplate code. As an example, we can use it to make queries such as "increase all the values of type *Int* by 10 in this data structure":

```
over (types @Int) (+10) (Left (0, False, Just 20))
```

resulting in *Left* (10, *False*, *Just* 30). Specifying how to locate all the *Int* values by hand would be a menial task, and the library can work it out for us.

Here we show how unsaturated type families can be used to reduce the volume of boilerplate code in the implementation of the library itself. We combine our extension with some of Haskell's unique features to devise a powerful *type-level generic programming framework*.

The payoff is substantial: using unsaturated type families allows us to reduce the size of the type-level code in the library *by a factor of five*. Moreover, the code is much higher-level and it is now easier to see what the various data structure traversals do. A key additional benefit is that they remain correct even if the underlying generic representation were to be extended with new constructors.

6.1 The Old Way

The generic-lens library uses type-level programming to perform a *compile-time* traversal over the shape of the data type, and generates optimised code that only accesses the pertinent parts of a data structure at runtime.

To achieve this, the library defines several queries over a generic structure that is available at the type-level. As an example, *HasCtor ctor f* uses the Haskell 'generics' library [Magalhães et al. 2010] to traverse a generic tree *f* and return *'True* if the type contains a constructor named *ctor*, *'False* otherwise:

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

Note that this uses non-linear patterns to check that the constructor symbol being searched for (*ctor*) is matched within the tree.

It is not essential to understand the details of the generics library, but we give a brief summary. The sum (:+:) represents the choice between two constructors, and product (:×:) represents 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 optionally field names) about the nodes. These representation types can automatically be derived for any algebraic data type, and the rest is taken care of by the generic-lens library.

Another query function in generic-lens is *HasField*, which returns *'True* if and only if *f* contains a field named *field* (recall that record types in Haskell have named fields):

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

There are many more type families along these lines, each traversing the generic tree to extract some information of interest.

We can see that both *HasCtor* and *HasField* are rather sizeable. After all, they handle all cases one-by-one, and recurse when appropriate. What's worse, they are almost identical, the only difference being the termination conditions. It is somewhat ironic that a library which was designed to eliminate boilerplate code itself contains a lot of boilerplate!

6.2 The New Way

We now show how the `UnsaturatedTypeFamilies` extension can be used to define a type-level generic programming framework to describe traversals in a more concise manner. We then show how to implement type families such as those above as one-liners.

The Scrap Your Boilerplate (SYB) [Lämmel and Peyton Jones 2003] library uses type equality tests to identify the relevant parts of data structures. We borrow this strategy and use the same interface for our type-level generic programming framework.

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

```
type family Everywhere (f :: b →^m b) (st :: a) :: a where
  Everywhere f (st :: b)    = f st
  Everywhere f (st x :: a) = (Everywhere f st) (Everywhere f x)
  Everywhere f (st :: a)    = st
```

Everywhere is already very powerful. For example, we can replicate the query from above[7]:

```
Everywhere (Add 10) ('Left '(0, 'False, 'Just 20))
>  'Left '(10, 'False, 'Just 30)
```

How was this so easy? We made use of some of Haskell's unique type system features. These are:

Kind-indexing Type families can intensionally inspect the kinds of their arguments when 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 is $st\ x$. This only matches when the input structure is a type application. This means that it will match $'Just\ 30$, but not $'False$, for example. In this case, *Everywhere* recurses into both sides, then reconstructs the application. We note that only *matchable* type constructor applications will match this pattern.

Note that the third equation matches anything not covered by the first two, by virtue of overlapping equations. Remark: application decomposition in the type system is precisely the reason why type families had to be fully saturated in the past.

The second combinator, *Gmap*, is similar to *Everywhere* in that it applies a function f 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.

```
type family Gmap (f :: b →ᵐ r) (st :: a) :: [r] where
  Gmap f (st :: a)    = '[f st]
  Gmap f (st x :: b) = Gmap f st ++ Gmap f x
  Gmap f (st :: b)    = '[]
```

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

```
type family Listify k (st :: a) :: [k] where
  Listify k st = Gmap (Id @k) st
```

Listify simply maps the identity type family *Id*, but instantiated to the kind $k \twoheadrightarrow k$ using type application in types [Nguyen 2018]. This means that *Gmap* will only pick up the types whose kinds are k, and ignore the others in the resulting list. (Notice that k is given as an argument to *Listify*, and used in its return kind: indeed, the type system is dependently kinded, with the $\star :: \star$ axiom [Weirich et al. 2013].) For example, we can query all the names (types of kind *Symbol*) that appear in the definiton of the *Maybe* type by:

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

where *Rep* is the type family that returns the generic representation of an algebraic data type. With our generic framework now in place, we can finally revisit the *HasField* and *HasCtor* functions.

```
type family HasCtor₂ ctor f where
  HasCtor₂ ctor f = Foldl (||) 'False (Gmap ((==) ('MetaCons ctor)) f)
```

[7] In this case, we use the type-level equivalents of the constructors, which are available thanks to promotion [Yorgey et al. 2012]

We map the function ((==) '*MetaCons ctor*) over the structure. This implicitly selects only values of kind *Meta*, and returns '*True* for the constructors called *ctor*. We then fold the result with the (||) function, defaulting to *False* in case the type had no constructors – in that case *Gmap* will return an empty list. This results in '*True* if any of the constructors were called *ctor*. *Foldl* is simply the value-level *foldl* function lifted to the type-level:

$$\begin{aligned}&\textbf{type family } \mathit{Foldl}\ (f :: b \to^m a \to^n b)\ (z :: b)\ (xs :: [a]) :: b \textbf{ where}\\ &\quad \mathit{Foldl}\ f\ z\ '[\,] = z\\ &\quad \mathit{Foldl}\ f\ z\ (x\ ': xs) = \mathit{Foldl}\ f\ (f\ z\ x)\ xs\end{aligned}$$

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

$$\begin{aligned}&\textbf{type family } \mathit{HasField}_2\ \mathit{field}\ f \textbf{ where}\\ &\quad \mathit{HasField}_2\ \mathit{field}\ f = \mathit{Foldl}\ (||)\ '\mathit{False}\ (\mathit{Gmap}\ ((==)\ ('\mathit{MetaSel}\ ('\mathit{Just}\ \mathit{field}))))\ f)\end{aligned}$$

To conclude, we have seen how a large class of type-level traversal schemes can be unified into a small set of combinators. In other dependently typed programming languages, this problem is traditionally solved by defining operations on a closed universe that can be interpreted into a type [Altenkirch et al. 2006]. This is required because $\star$ is not inductively defined.

Instead, type families in Haskell allow pattern matching on syntactic properties of elements of $\star$. Namely, matching on application forms of unknown type constructors together with kind-indexing allowed us to write recursive definitions such as *Everywhere* and *Gmap* over *all* types, without having to assume a recursion principle for the underlying set.

7 RELATED WORK

Type families were first introduced into Haskell as associated type families [Chakravarty et al. 2005] and several extensions have since been added, most notably closed and injective type families [Eisenberg et al. 2014; Stolarek et al. 2015]. From the perspective of this paper the key point is that instance declarations introduce axioms (Section 4.3) regardless, and the only differences between the different family types is additional typing information that accrues from the family's definition, e.g. associated class constraints and argument injectivity. None of these impact our implementation.

7.1 Previous Work on System F_C

In [Weirich et al. 2011] the coercion decomposition rules are changed to work only between known type constructors. Instead of the **left** and **right** rule, which work on any equalities of the form $f\ a \sim g\ b$, a restricted *nth* rule is introduced, which projects out the equality of the "n"th argument of $T\ \overline{a} \sim T\ \overline{b}$, where T is an injective type constant. While this system allows unsaturated type functions, it weakens type inference by not allowing decomposition of given equalities.[8]

Haskell's template metaprogramming facilities [Sheard and Peyton Jones 2002] have been used to generate each possible partial application of a given type family [Eisenberg and Stolarek 2014]. This uses *defunctionalisation* [Reynolds 1972], a well established technique for translating higher-order programs into a first-order setting. The defunctionalisation symbols are distinguished in the kind system, which served as direct inspiration for our work. We improve the ergonomics by extending the type system with first-class support for unsaturated type families.

[8]In fact, in anticipation of this feature, the **left** and **right** coercion forms were briefly removed from GHC, only to be added back in a subsequent release, as type inference suffered. https://gitlab.haskell.org/ghc/ghc/issues/7205

7.2 Dependent Haskell

The various type system extensions as seen in GHC have been moving Haskell closer and closer to supporting full-spectrum dependent types. Dependent Haskell will allow ordinary term-level functions in types [Weirich et al. 2017]. However, getting there poses a unique challenge: backwards compatibility. Programs that compile today should also compile in Dependent Haskell and type inference should not be compromised.

Type inference in the context of dependent types has been investigated in [Gundry 2013]. They maintain a phase distinction between terms and types, with a notion of *shared* functions that are usable in both settings. Shared functions must be fully saturated to maintain the desired injectivity and generativity properties.

This restriction is lifted in [Eisenberg 2016] by distinguishing between matchable and unmatchable functions, in a fully dependently typed calculus which replaces System F_C. We describe the feature in the context of type families and System F_C, so it is readily applicable to GHC today. Our treatment of matchability polymorphism is novel, which leads to more predictable type inference than the subsumption relationship proposed in [Eisenberg 2016].

7.3 Full-Spectrum Dependently Typed Languages

In languages like Agda [Norell 2007] and Idris [Brady 2013] that support full-spectrum dependent types, partial application of type functions is standard practice. These systems do not assume injectivity of unknown constructors, so avoid the problem of unsound decomposition. In fact, type constructor injectivity is generally problematic in the presence of classical axioms such as the law of excluded middle, so even known type constructors are not injective in proof systems [Hur 2010].

8 CONCLUSIONS AND FUTURE WORK

Our implementation of unsaturated type families, which is an extension to GHC, is non-invasive in the sense that it requires no significant change of GHC's existing constraint solving algorithm. Existing programs that compile under GHC also compile with our extension. As a demonstration of the robustness of our implementation, our fork of GHC can bootstrap itself, and all the examples in this paper are valid type-checked programs.

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 work, in anticipation of them becoming redundant in Dependent Haskell.

Our approach to type-level generic programming is somewhat unique to Haskell as it involves the interaction of several of Haskell's type system features that are not present in mainstream dependently-typed languages, namely intensional type analysis and the fact that application decomposition is possible on polymorphic type constructors.

This work addresses the tension between rich type programming and good (and simple) type inference. As our implementation in GHC shows, it is possible to have both higher-order type functions and a simple first-order unification algorithm thanks to the matchability information which guides the constraint solver. The virtues of staying in a first-order unification world are not limited to Haskell, and this work is equally applicable to other languages, such as Scala, which compromises type inference to support type lambdas, and PureScript, which does not allow higher-order type programming in order to maintain good type inference.

8.1 Type Lambdas

Type lambdas are not yet supported and introducing them will be non-trivial, as they open up the possibility of unification problems where the solution can have binding structure. Higher-order unification, in general, is undecidable [Huet 1973].

Many systems implement a decidable subset, such as Miller's pattern fragment [Miller 1992], where higher-order metavariables must be applied to distinct bound variables. This can be improved upon and matchability information can help here. As an example, suppose we want to express that the composition of two functors is itself a functor:

$$\textbf{instance}\ (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow \mathit{Functor}\ (\lambda x \to f\ (g\ x))\ \textbf{where}$$
$$\quad \mathit{fmap} :: (a \to b) \to f\ (g\ a) \to f\ (g\ b)$$

Notice that in the lambda, the bound variable x appears in a matchable position, because both f and g are matchable. Now suppose we call *fmap* with a function of type $a \to b$, and an argument of type $\mathit{Maybe}\ [a]$. Which instance should be picked? We need to solve for β in the equality:

$$\beta\ a \sim \mathit{Maybe}\ [a]$$

The solution is $\beta := \lambda x \to \mathit{Maybe}\ [x]$, which unifies with our instance above. What if the function had type $[a] \to [b]$ instead?

$$\gamma\ [a] \sim \mathit{Maybe}\ [a]$$

This can be solved by assigning $\gamma := \mathit{Maybe}$, so the instance for *Maybe* can be picked.

Note that supporting this will require a modified notion of generativity where the arguments to type functions have to match, *viz.* $f\ a \sim g\ a \Rightarrow f \sim g$.

8.2 Matchability Inference

The matchability defaulting strategy described in Section 5.1 is incomplete: there are some well-typed programs that it doesn't accept. Consider the following:

$$\mathit{nested} :: a\ b \sim c\ \mathit{Id} \Rightarrow b\ \mathit{Bool}$$
$$\mathit{nested} = \mathit{False}$$

What should the inferred matchabilities of a, b, and c be? Defaulting all of them to be matchable enables the decomposition of the equality, and by doing so we learn that $b \sim \mathit{Id}$. However, we just defaulted the kind of b to be matchable. This does not threaten type safety, but it means that the caller needs to instantiate b with a matchable type that is equal to *Id*. Of course, no such type exists, so the function can never be called!

We can, of course, fix the above problem by manually declaring b's kind:

$$\mathit{nested} :: \forall\ (b :: \star \twoheadrightarrow \star).\ a\ b \sim c\ \mathit{Id} \Rightarrow b\ \mathit{Bool}$$

but this seems unfortunate. Could we do better?

The issue is that defaulting everything is too eager. For example, if we were to default only a and b to matchable then we would enable new interactions in the constraint solver, namely deducing that $b \sim \mathit{Id}$ and thus b is unmatchable.

With this in mind we have flirted with a more elaborate inference algorithm that recognises that b's matchability is constrained by that of a and c, and defers defaulting b until a and c are resolved.

This type of situation might be quite rare in practice, so the complexity of a complete inference algorithm might not pay its way. Of course, this is just speculation, and time will tell whether the simple method is sufficient, or overly restrictive. The good news is that if it turns out to be the latter, type inference can be extended in a backwards-compatible way, because a more sophisticated algorithm would just accept more programs.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and everyone else who provided valuable feedback on earlier versions of this manuscript, in particular Richard Eisenberg, Sylvain Henry and John Ericson. Thanks also to Nicolas Wu, Will Jones, and Ryan Scott for their insightful comments and to Matthew Pickering for helping to package the artefact.

REFERENCES

Thorsten Altenkirch, Conor McBride, and Peter Morris. 2006. Generic programming with dependent types. In *International Spring School on Datatype-Generic Programming*. Springer, 209–257.

Maximilian C. Bolingbroke. 2011. Constraint Kinds for GHC. http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/

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

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

Richard A Eisenberg. 2016. *Dependent types in Haskell: Theory and Practice*. Ph.D. Dissertation. University of Pennsylvania.

Richard A Eisenberg and Simon Peyton Jones. 2017. Levity polymorphism. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 525–539.

Richard A. Eisenberg and Jan Stolarek. 2014. Promoting Functions to Type Families in Haskell. In *ACM SIGPLAN Haskell Symposium*.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 671–683. https://doi.org/10.1145/2535838.2535856

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *ESOP*.

GHC. 2019. GHC User Manual on Type Families. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#type-families Accessed: 2019-05-31.

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types*. Ph.D. Dissertation. University of Strathclyde.

Gerard P Huet. 1973. The undecidability of unification in third order logic. *Information and Control* 22, 3 (1973), 257–267.

Chung-Kil Hur. 2010. Agda with the excluded middle is inconsistent? https://coq-club.inria.narkive.com/iDuSeltD/agda-with-the-excluded-middle-is-inconsistent. Accessed: 2019-03-01.

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

Csongor Kiss, Matthew Pickering, and Nicolas Wu. 2018. Generic Deriving of Generic Traversals. In *International Conference on Functional Programming (ICFP '18)*. ACM. arXiv:arXiv:1805.06798

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Types in Languages Design and Implementation*. ACM Press, New York, NY, USA, 26–37. https://doi.org/10.1145/640136.604179

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A Generic Deriving Mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1863523.1863529

Dale Miller. 1992. Unification under a mixed prefix. *Journal of symbolic computation* 14, 4 (1992), 321–358.

My Nguyen. 2018. Type-level visible type application. Talk, Haskell Implementors Workshop, St. Louis, MO, United States.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

John Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *ACM Annual Conference*.

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

Vincent Simonet and François Pottier. 2007. A Constraint-based Approach to Guarded Algebraic Data Types. *ACM Trans. Program. Lang. Syst.* 29, 1, Article 1 (Jan. 2007). https://doi.org/10.1145/1180475.1180476

Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. 2015. Injective Type Families for Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 118–128. https://doi.org/10.1145/2804302.2804314

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

Anish Tondwalkar. 2018. Popularity of Haskell Extensions. https://gist.github.com/atondwal/ee869b951b5cf9b6653f7deda0b7dbd8. Accessed: 2019-02-24.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of Functional Programming* 21, 4-5 (Sept. 2011).

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with Explicit Kind Equality. In *International Conference on Functional Programming (ICFP '13)*. ACM.

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. 2011. Generative type abstraction and type-level computation. In *POPL*. ACM.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Principles of Programming Languages (POPL '03)*. ACM.

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