Utrecht University

Thesis Report

Department of Information and Computing Sciences

Implementing Type Families in Helium

A study on improving Type Error Diagnosis for Type Families in Haskell

---

Author:
N. Kwadijk, BSc
n.kwadijk@uu.nl
5911699

Supervisors:
prof. dr. J. Hage
dr. W.S. Swierstra

July 2022

Universiteit Utrecht

Abstract

Type families in Haskell allow the programmer to be more expressive on the type level. Type inference on the other hand becomes more complicated and the quality of error messages suffers. With the introduction of Rhodium to Helium, we are able to introduce type families to the compiler and use its techniques to improve error messages. To this end, we designed a reduction trace that traces the applications of type families throughout the type inference process. Furthermore, we designed an error message structure and hints to help the programmer diagnose the error. Hints were designed to help with permutation mistakes, warn the programmer for apartness and suggest injectivity annotations when needed. The techniques for error message improvement in Helium are certainly ready to include type families. However, there is some terrain to win with respect to comprehension of type error messages. Last but not least, the reduction trace is not complete in the sense that it is not able to obtain a trace for all programs.

Contents

Chapter 1

Introduction

Haskell is a pure, lazy, statically typed, functional programming language that is designed to be suitable for teaching, research, and applications [18]. These different use cases resulted in many features that were implemented for Haskell over time. While a broad feature set increases the use cases for the language, there is one drawback: the more features a language has, the more sources an error can originate from. As a result, error messages often can become confusing for programmers. Not only does this happen to programmers new to the language, the more advanced features in Haskell frequently baffle experienced programmers as well. One such advanced feature is Type Families [29] (Section 2), that allow the programmer to define functions on the type level. This feature opens up many extra ways to extend type correctness and allows the programmer to provide a high degree of certainty with respect to the correctness of a program.

Even though this feature is not meant to be used by novices in Haskell, even experienced programmers had to learn to work with it. It is therefore important that errors, and more specifically *type errors*, are clear and to the point. In the standard Haskell compiler, the Glasgow Haskell Compiler (GHC), this is often a point of concern. Error messages tend to be confusing and deviate the programmer from the exact source of the error. Currently, Helium [15] (Section 3.1) is a Haskell compiler that is specialized in type error diagnosis. It uses a number of techniques to improve it. The compiler is not as extensive as GHC and only covers a subset of the features that it offers. Recently, the type error diagnosis techniques of Helium have been extended to work with one of the more advanced features of Haskell: GADTs [26]. A new type inference engine, Rhodium [4], has been designed to make the type error diagnosis techniques in Helium compatible with the more advanced features that GHC implements. Of those features, GADTs are currently the only one implemented.

Upon Rhodium, we have implemented Type Families in Helium. Using the type error diagnosis techniques from Helium, we have designed ways to improve the type error diagnosis for Type Families in Helium with respect to GHC. To this end, we extended the parser, static checks and type inference of Helium to support Type Families. Furthermore, we designed a *Reduction Trace* that tracks how types using Type Families are transformed throughout the type inference process. Finally, we designed and implemented new *heuristics* for Helium that allow further reasoning over type errors concerning Type Families.

1.1 Relevance

Providing understandable type error messages in functional languages is hard. Especially when the type system they work with becomes more and more advanced. This also applies to type families, which introduce function applications to the type system, making it more complex. Furthermore, type families introduce many rules to ensure termination and correct application. This introduces confusing situations for the programmer. One such rule is the *apartness rule* (Section 4.4.1).

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

loop :: Loop [c]
loop = 'X'
```

At first glance, a (novice) programmer would expect that `Loop [c]` would reduce to `c` by first applying the first instance and then applying the second. The second instance is, however, never applied and GHC gives the following error message:

```
main.hs:206:8: error:
    • Couldn't match type ‘Loop c’ with ‘Char’
      Expected type: Loop [c]
        Actual type: Char
    • In the expression: 'X'
      In an equation for ‘loop’: loop = 'X'
    • Relevant bindings include
        loop :: Loop [c] (bound at main.hs:206:1)
    |
206 | loop = 'X'
    |
```

`Loop c` cannot be reduced because `c` can be instantiated as a list type in which case the first instance would be applied. The apartness rule states that we must be completely sure that another instance could never be applied. For type families, we only need a very simple example to cause the type inferer in GHC to give a confusing error.

```
type family Foo a
type instance Foo Int = Float

wut :: Foo Float
wut = 6
```

Notice that the type `Foo Float` is not reducible by the type family. We thus expect that `Foo Float` cannot be matched with the type `Int` from `6`. GHC, on the other hand, provides the following error message:

```
main.hs:179:7: error:
• No instance for (Num (Foo Float)) arising from the literal '6'
• In the expression: 6
    In an equation for 'wut': wut = 6
    |
179 | wut = 6
    |
```

GHC expects an instance of `Num` for `Foo Float`, which obviously is not there. This error does not provide the main cause of the type error and instead takes the focus of the programmer off of it. In this thesis, we use the techniques in Helium to clarify confusing type family rules and always provide the correct source of the error.

1.2 Document overview

In chapter 2 we first provide a short introduction to Type Families to provide an idea of how they emerged and what they are used for. In chapter 3 we discuss the current state of research with regards to existing literature. This section excludes a detailed discussion on the implementation of Type Families as this is discussed in chapter 4. Here, we focus on design choices made for Type Families in the GHC compiler and the consequences for their expressiveness. Chapter 5 discusses type error messages that GHC provides when certain errors are made concerning type families. We discuss what makes them confusing and how they could be improved. Using the context gathered in these chapters, we introduce our research questions in chapter 6. The next chapters introduce and discuss the work that has been done on improving type error diagnosis for type families to answer the research questions. Chapter 8 introduces the reduction trace implemented to keep track of type family applications. Chapter 9 introduces the main type family heuristic designed to provide better error messages for left-to-right type family reductions. Next, chapter 10 discusses the heuristics implemented to provide better error messages regarding injective type families. We finish the implementation section with a comparison of the error messages in Helium with the error messages in GHC. This is done in chapter 11. Last but not least, we conclude our work in chapter 12 and provide future work that can be done to improve on this work in chapter 13.

Chapter 2

Preliminaries: Type Families

In Haskell, a type family is a synonym for a type level function. Type level functions allow the programmer to write entire programs at the level of types [29] and it allows the type inference system to infer different types based on different arguments given to the type function. The idea is similar to a value-level function. In this section, we discuss the different ways in which type-level functions can be defined in Haskell and how they differ from each other. Readers with experience in type families could probably skip this section. Knowledge of type classes in Haskell is assumed.

2.1 Functional Dependencies

Type level functions first arose with the introduction of *Functional Dependencies* [20]. A functional dependency is an extension to multi-parameter type classes that allow the programmer to define dependencies between different parameters. Let us use the following example.

```
class Collects e ce where
    empty :: ce
    insert :: e -> ce -> ce
    member :: e -> ce -> Bool
```

At first glance, this class should not pose any problems and we should be able to create some nice instances for it. However, a problem arises when considering `empty`. It has the following *ambiguous* type:

```
empty :: Collects e ce => ce
```

With ambiguous we mean that there is a type variable e on the left side of => but not on the right. This results in the situation where the type inference system is not able to assign a type to e and is thus not able to select the right type class instance to use. Functional dependencies can solve this problem by redefining the class definition as follows.

```
class Collects e ce | ce -> e where
    empty :: ce
    insert :: e -> ce -> ce
    member :: e -> ce -> Bool
```

Here, `ce -> e` means that "*ce* uniquely determines *e*". As a result, empty now becomes unambiguous as the type inference system can now assign a type to e when ce is known. It is the job of the type system to make sure that the dependency holds. Notice that this dependency can be seen as a nameless function that, given a type for ce, returns a type for e. Thus, we see here the first form of a type-level function.

2.2 Associated Type Synonyms

Associated type synonyms [5] take the idea of functional dependencies and rewrite the syntax in such a way that an actual type-level function arises. The definition of *Collects* can now be rewritten using associated type synonyms as follows.

```
class Collects ce where
    type Elem ce
    empty :: ce
    insert :: Elem ce -> ce -> ce
    member :: Elem ce -> ce -> Bool
```

Notice that the element parameter *e* from the functional dependency is now encoded as *Elem ce* which can be instantiated differently per class instance. Let us take the following instance for lists.

```
instance Collects [a] where
    type Elem [a] = a
    empty = []
    insert x xs = (x:xs)
    member = elem
```

Note that the element for [*a*] is defined as *a*. As a result, the type inference system will substitute the type *Elem* [*a*] with *a* in functions like *insert* and *member*. Expressively, the definitions using functional dependencies and associated type synonyms are the same. It is difficult to decide which one is better as both options have their use cases.

2.3 Open Type Families

Open type families [29] take the idea of associated type synonyms and decouple it from type classes to make them stand-alone. The main advantage here is that the type-level functions are now orthogonal to other constructs. The syntax is quite straightforward. One defines a family and is then able to write instances for it. For example, let us take the following type-level function *Sum*

```
type family Sum n m
type instance Sum Zero x = x
type instance Sum (Succ x) y = Succ (Sum x y)
```

which is similar to what one can write on value level. The most interesting use case for open, independent type families like *Sum* is in combination with Generalized Algebraic Datatypes (GADTs). A well-known example is the definition of a vector, as follows.

```
data Nat = Z | S Nat

data Vector el len where
    Nil :: Vector el Z
    Cons :: el -> Vector el len -> Vector el (S len)
```

Nat defines the natural number, which can be either zero or a successor. The Vector is defined with its element type `el` and the length it has, `len`. `Nil` naturally has length Zero. `Cons` takes a new element (of type `el`) and a vector and returns a new vector in which the length is increased. We can write functions for such a vector. For example, we may concatenate two of them.

```
vconcat Nil l = l
vconcat (Cons x xs) ys = Cons x (vconcat xs ys)
```

To be able to define a correct type for this function, we need type-level computation as we need to find the new length of the vector. For this we can use the *Sum* type family we defined earlier.

```
vconcat :: Vector e n -> e Vector e m -> Vector e (Sum n m)
```

Notice that the result type now sums the two lengths of the argument vectors. The type family definition then ensures that the type inference process simplifies this to a simple type of type *Nat*.

2.4 Closed Type Families

Closed type families [8] allow the programmer to strictly group the instances of a type family together in one statement. Outside of that statement, no other instances can be defined. This has one major advantage in the sense that instances are allowed to overlap. As a result, a type family can become more expressive. This is discussed in more detail in section 4. The syntax is quite straightforward. Consider the example *Append*.

```
type family Append xs ys where
    Append '[] ys = ys
    Append (x:xs) ys = x : Append xs ys
```

Note that, outside of the where clause, no other definitions of *Append* are allowed.

2.5 Injective Type Families

Injective Type Families [34] allow the programmer to annotate whether a type family has injective properties or not. Due to reasons explained in section 4, the type inference system does not infer if a family has such properties but instantly dismisses any situation that requires it to be so. An injective type family is able to infer some of its arguments based on the result type. Let us take the following contrived example.

```
type family F a b = r | r -> b where
    F Int Int = Bool
    F Bool Bool = Int
    F Int Bool = Int
```

We say that type family F is injective in its second argument b as specified by its annotation. This means that the right-hand sides (rhs) of its instances determine only the second arguments on the left-hand side (lhs). F is not injective in its first argument as its rhs does not determine it. *Int* as result can be the result of *Int* or *Bool* as the first argument. The usefulness of injective type families will be discussed in section 4.

2.6 Example usage of Type Families

Type Families may be used in many cases to enrich the type system. They are often used in combination with GADTs because of the complexity of those data structures. We show this use case through an example where we define a *heterogenerous list*.

```
type HList :: [Type] -> Type
data HList xs where
  HNil :: HList '[ ]
  (:&) :: x -> HList xs -> HList (x : xs)

h1 :: HList [Char, Bool]
h1 = 'x' :& False :& HNil

happend :: HList xs -> HList ys -> HList ??
happend HNil ys = ys
happend (x :& xs) ys = x :& happend xs ys
```

A heterogeneous list is a list that can contain values of different types and is constructed using a GADT. It has a Nil case, which represents the empty list, and a Con case with which we may add new elements to an existing list. `h1` shows an example of how a heterogeneous list may be constructed. This example becomes interesting when we want to define a function to append two heterogeneous lists: `happend`. To be able to construct the result type, we must somehow append `HList xs` and `HList ys`. To this end, we may define a type family:

```haskell
type Append :: forall a. [a] -> [a] -> [a]
type family Append xs ys where
  Append '[ ]    ys = ys
  Append (x:xs) ys = x : Append xs ys
```

The type family performs exactly the same computation that `happend` does, only on the type level instead of on the value level. We may now insert the type family in the type of `happend` to compute the correct result type at compile time.

```haskell
happend :: HList xs -> HList ys -> HList (Append xs ys)
happend HNil ys = ys
happend (x :& xs) ys = x :& happend xs ys
```

Chapter 3

Literature review

3.1 Helium

Helium [15] is a user-friendly compiler designed especially for excellent support during learning the functional language Haskell. This is achieved by providing high-quality error messages for every compile stage. This has been proclaimed by, for example, Gerdes et al. [9]

> ”Helium gives excellent syntax-error and type error messages”

The quality of the error messages is achieved by a number of techniques, most notably the TOP framework (Section 3.2.4) and later Rhodium (Section 3.2.5) in combination with a large number of heuristics. With Rhodium, the Helium compiler is able to compile most of the Haskell 2010 standard including Generalized Algebraic Data Types (GADTs). The compiler, in its current state, is thus not able to support many of the language extensions that exist for Haskell today. Hage [11] shows that many of these extensions are widely used, indicating the importance of adding support for them to Helium. At the time of writing, Helium is in the process of being converted to support LLVM compilation instead of LVM [23] with which it was originally built. Both LLVM and LVM currently support all the language constructs that Helium supports except for GADTs, which are supported up to the type inference process with the use of Rhodium.

3.2 Type Checking and Inference

Type checking and inferring is the process of respectively validating and inferring the types of a program. In Haskell and many other languages, these processes are usually combined. Provided types in type signatures are checked while other, non-specified types are inferred. Literature has devised many techniques, each with its advantages and disadvantages. This section discusses some of the most well-known and most important for this document.

3.2.1 Algorithms W and M

Algorithm W, as devised by Damas et al. [6], is an inference algorithm for the Hindley-Milner type system as originally constructed by Hindley [17]. It is one of the most widely known type inference algorithm. The algorithm itself is not very relevant for this thesis but some of its limitations are. The most important of them all is that the algorithm has a bias in the order in which it checks and infers types. For algorithm W, this happens in bottom-up order. As a result, it always reports the location of the first inconsistency it encounters as the error. In many cases, this location may not be as relevant or as useful for the programmer as other locations that are related to the error. Algorithm W was not designed with type error diagnosis in mind and thus shows this *left to right bias.* Algorithm W also tends to report errors quite late. As an alternative algorithm to W, algorithm M [22] is a top-down algorithm for checking and inference of the Hindley-Milner type system. M is interesting because its top-down approach may result in different errors than algorithm W when checking the same program. Algorithm M tends, as directly opposed to W, to report errors in a quite early location. However, it still suffers from a bias. It is hard, or even impossible, to decide which algorithm is better. M may provide a more precise error location. W, on the other hand, may give more context to the error as it considers more constraints before exiting.

3.2.2 Constrained-based Type Inference

Constraint-based type inference is the process of generating type constraints from a given program and then solving those constraints to obtain a solution. At the end of the process, the solution represents the type of the whole program. The process of constraint-based type inference is described by Aiken and Wimmers [1]. They state that, if a solution is found, the program that generated the constraints is type correct. If not, the program contains an error and the compilation should exit.

In general, there are two ways that a set of constraints can be made inconsistent, as explained by Zhang et al [40]. The first is when there are incorrect constraints present in the constraint set. This could happen when a function is implemented incorrectly which results in spawning an incompatible constraint. Zhang et al. call these *incorrect hypotheses*. An example of an incorrect constraint is $Int \sim Bool$ which results in a type error as it can never be satisfied. The second way a constraint set can be inconsistent is by the absence of constraints, or as Zhang et al. call it: *missing hypotheses*. In this case, the programmer omitted necessary assumptions for the type inference system to continue. One such example is a missing class predicate, like $Eq\ a$. This happens in the following function.

```
isEqualToFive :: a -> Bool
isEqualToFive = (==) 5
```

$(==)$ requires a to be an instance of $Eq\ a$ but the constraint is missing in the type signature.

Many constrained-based type inference systems have been devised in literature. For this thesis, we discuss the three most relevant ones: OutsideIn(X), which is used in the Glasgow Haskell Compiler (GHC) for Haskell; TOP, which is the inference system used in the current master branch of Helium; and Rhodium, an adaptation of the OutsideIn(X) framework designed for better type error diagnosis. The next sections discuss them in detail.

3.2.3 OutsideIn(X)

OutsideIn(X), as designed by Vytiniotis et al. [36], is a modular constraint-based type inference system that supports local assumptions. The X resembles a constraint system that can be specified by the language developers and is thus extensible and adaptable. The main advantage that this feature brings, is that new language constructs and other features can be added later on. The system supports local assumptions, which is crucial for constructs like GADTs. Each pattern match over a GADT constructor may impose constraints that are only viable inside that pattern match and are therefore not allowed to interact with any constraint that is part of another pattern match.

In its solving process, the solver distinguishes *given* and *wanted* constraints. Given constraints represent existing knowledge, like a type signature. Wanted constraints represent the constraints that need to be verified or inferred. Furthermore, it keeps track of whether type variables are *touchable* or not. A touchable variable is allowed to be unified with a type. For example, in cases where a type variable is polymorphic, this should not be allowed.

Because OutsideIn(X) is a constraint-based solver, the compiler for which it is implemented should gather the constraints. OutsideIn then simplifies the constraints by looping over the following four simplification stages:

- **Canonicalization**. This step obtains one constraint and tries to simplify the constraint to an equivalent form. For example, $a \to b \sim c \to d$ becomes $a \sim c \wedge b \sim d$ and $Bool \sim Bool$ becomes ϵ. If a constraint cannot be simplified, an error is returned. An example of such a constraint is $Int \sim Bool$. The canonicalization rule is also able to return new touchables and a possible substitution, may the constraint impose so.

- **Interaction** The interaction rule allows two constraints, that are either both wanted or both given, to interact. The result of this interaction is a set of new constraints. As an example $a \sim b \wedge c \sim a \to a$ becomes $a \sim b \wedge c \sim b \to b$ where we see that the fact that a is equal to b is applied in the other constraint. Notice that we also return the original constraint. This is needed because $a \sim b$ may need to interact with other constraints as well.

- **Simplification** The simplification step takes one given constraint and one wanted constraint and allows the given constraint to interact *in one way* with the wanted constraint. It is important that the given constrain is not modified and that it is not returned after the step. Apart from that, the rule is similar to the interaction rule. In this stage, existing knowledge is applied to the wanted constraint system.

- **Top-level react** The top-level react rule allows the solver to apply top-level definitions (*axioms*) to an atomic constraint. This is needed to simplify class constraints or type family constraints. Constraints that need to be simplified by this step are, for example, $F\ Bool \sim Int$ or $Eq\ Int$ where F is a type family and Eq is a type class.

The exact implementation of these rules is determined by the parametric X and depends on the exact constraint system used. The complete solver can be assigned the following type:

$$\mathcal{Q}; \mathbb{Q}_{given}; \bar{\alpha}_{tch} \mapsto^{solve} C_{wanted} \rightsquigarrow Q_{residual}; \theta \tag{3.1}$$

The inputs of the solver thus are:

- the axiom set $\mathcal{Q}$,
- the set of given constraints obtained from the program $\mathbb{Q}_{given}$,
- the touchable variables $\bar{\alpha}_{tch}$ that the solver is allowed to unify and
- the set of wanted constraints C_{wanted} that the solver is requested to solve.

The output of the solver is the set of residual constraints $Q_{residual}$ that the solver could not solve and a substitution θ that contains a substitution for a subset of $\bar{\alpha}_{tch}$. The judgment $\mapsto^{solve}$ represents the solving of C_{wanted} using the set of touchables, axioms, and the given constraints. The reason why the solver may return residual constraints is that local constraints may not yet be fully satisfied after the solver is done simplifying the local constraint domain. Instead of making assumptions about the constraints, it just passes them back, hoping that they will be solved in a future iteration.

3.2.4 The TOP Framework

TOP is a type inference framework written by Heeren [16]. TOP is a constraint-based type checker and inferencer that is used in the Helium compiler. TOP has been built with type error diagnosis in mind, which is also its main advantage. The framework achieves this by representing the constraint system in a *type graph* which allows the framework to traverse the complete constraint system in an unbiased way when considering the location of the error. Additionally, it implements *type class directives* (Section 3.4.1) and *specialized error messages* (Section 3.4.2) to enable a programmer to specify directed error messages for Domain Specific Languages (DSLs).

TOP allows for the usage of different constraint solvers and is able to emulate, for example, Algorithms W and M. Through reordering constraints and the use of *heuristics*, the framework can return different error messages for the same programs. A *heuristic* is a function that can indicate the likelihood that a certain error has a specific cause. A heuristic is able to remove a constraint edge from the type graph to find whether the removal of that edge, fixes the constraint system.

In TOP, every constraint can be annotated with specific information. This information is used by the heuristic functions to improve their judgment on the likely cause of the type inconsistency. As adding new annotations to constraints is quite easy, compiler programmers can simply construct new heuristics so that new language features can benefit from the improved type error diagnosis that the TOP framework brings.

Because the type graphs of TOP are quite sophisticated, they are only created when a type error is found. The framework usually starts with a greedy solver after which a type graph is created if this solver fails.

Type graphs in TOP The type graphs in TOP consist of a set of edges and vertices. In a simple type graph, edges represent equalities between types (equality constraints) and vertices represent simple types, which are either constants or variables. Figure 3.1 shows an example of a simple type graph where an error occurs through the following constraint sequence:

$$Int \sim a \sim b \sim c \sim Bool$$

The error is formed by edges 1 to 4. These edges together form the so-called *error path*. Every edge in this path may be the cause of the error and should be considered as such. Edge 5 is not part of the error path as the equality $b \sim d$ has nothing to do with the error at hand.

Such a simple type graph can be extended to, for example, support type application. As a result, type applications like $\tau_1\ \tau_2$ are supported. Such an application is visually represented by square vertices

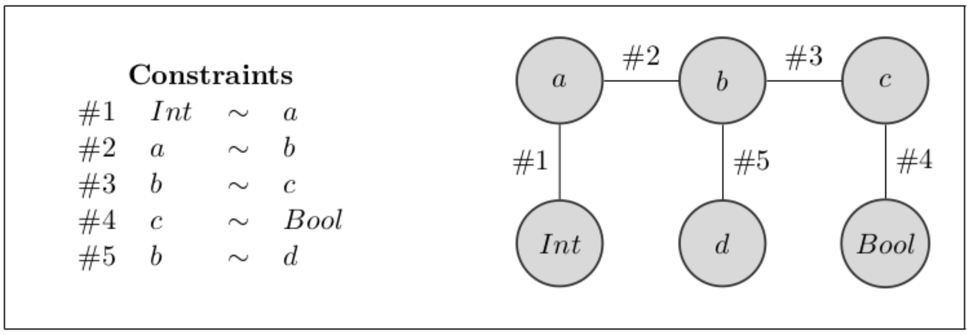

Figure 3.1: An example of a simple type graph (taken from [3]).

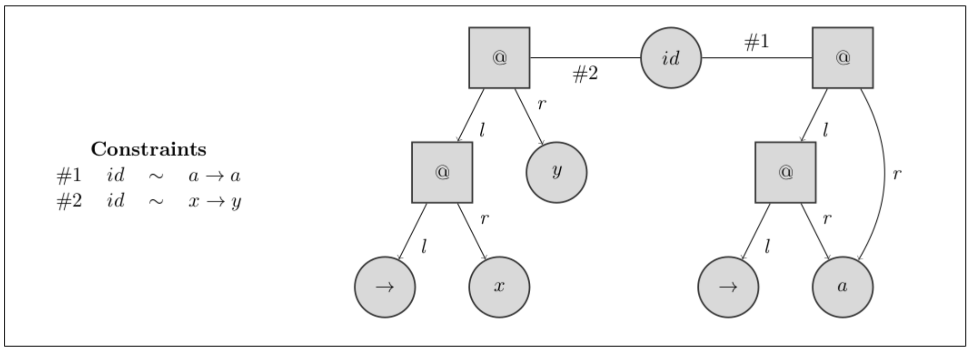

Figure 3.2: An example of an extended type graph (taken from [3]).

with an @ inside them. The extended type graph abandons the convention that all vertices represent types and that every edge represents an equality.

Figure 3.2 shows an example of an extended type graph. To left and right of *id*, we see how a type application is visualized for $a \to a$ and $x \to y$. The $\to$ can be seen as a function that takes two arguments. In the type graph, we notice that it is indeed visualized as a nested application where $\to$ is first applied to x and then $(x \to)$ is applied to y to form $x \to y$.

Inconsistencies in type graphs are found by the introduction of derived edges during type inference. These edges are created based on combinations or simplifications of existing equalities in the graph. For example, we could have the equalities $a \sim A$ and $a \sim B$. A derived edge created from these equalities would be $A \sim B$. As the constructors A and B are different, we would notice an inconsistency. An error would then be reported.

3.2.5 Rhodium

Rhodium is a type inference framework written by Burgers et al [4] and aims to bring the best of OutsideIn(X) and the TOP framework together. As TOP does not support local assumptions and type families, the Helium compiler was not able to deal with them. Rhodium extends the OutsideIn(X) framework to support type graphs and the accompanying heuristics that were written for Helium. Like OutsideIn(X), Rhodium supports the parametric X and assumes nothing about the constraint system.

The X written for Rhodium in Helium is heavily based on the X written in Cobalt [31], a small experimental type inference system written by Alejandro Serrano. The name Rhodium originates from the fact that the next element on the periodic table after Cobalt, is Rhodium. The X in Cobalt has support for type families implemented as far as the use of open type families. Type synonyms are regarded as very simple type families.

Rhodium itself implements the same simplification rules as OutsideIn(X). Its main difference is that, instead of working with sets of constraints, it works on a type graph. Based on the specified X, an initial type graph is built. After each simplification step, this graph is updated to include the newly obtained constraints. The type graphs in Rhodium are extended to work with local assumptions and with given and wanted constraints. In the graph, every constraint edge has a *priority* and a *grouping* assigned to it. The priority determines whether the constraint is given or wanted. The grouping of a constraint edge tells the inference system in what local domain it resides and thus with what other constraints it may interact.

In comparison to the solving algorithm of OutsideIn(X), Rhodium does not apply substitutions during the solving process. The reason for this is the fact that, for optimal type error diagnosis, we would want all original constraints to remain intact so that they still closely represent the original program. Applied substitutions would be hard to revert.

Type graphs in Rhodium As mentioned, type graphs in Rhodium are extended to hold priorities and groupings for vertices and edges due to local constraints. Next to the incorporation of type application, like in the type graphs for TOP, type graphs in Rhodium also support multiparameter type classes and type families. Type families are implemented using the application edge. Figure 3.3 shows a visualization of the implementation in a type graph. The edges are numbered from 0 to n. The zero edge is always

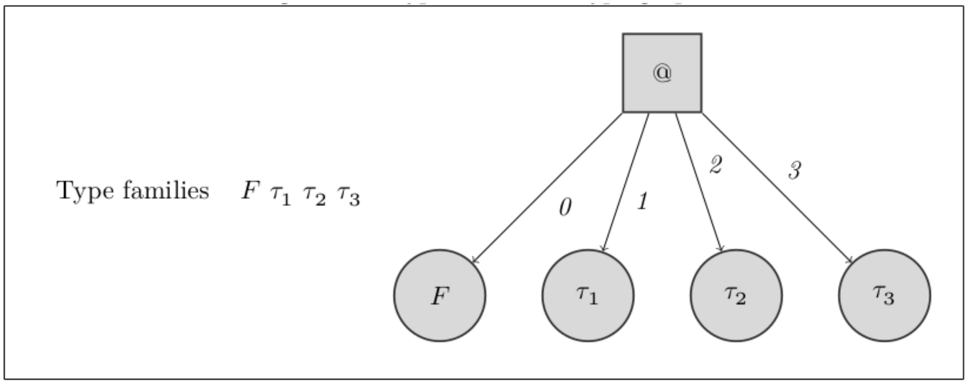

Figure 3.3: Type families in a type graph (taken from [3]).

linked to the vertex containing the family name F. The rest of the edges are linked to the argument types of the type family application. The numbers determine the order of the arguments.

3.3 Type Error Diagnosis

Type error diagnosis is the process of explaining why a type error occurred in a program and helping the programmer locate and fix the error. This process usually consists of three phases [32, 33]: blaming, reparation and explanation. Blaming is the phase that decides what parts of the program are responsible for the error. The reparation phase suggests ways in which the error can be repaired. This may happen by removing, changing, or adding a new constraint. Last but not least, the explanation phase is concerned with providing the error and possible (hints towards) solutions that could help the programmer repair the error. This phase explains why the error was thrown in the first place.

3.3.1 What are good errors?

Yang et al. [39] provide a manifesto that gives seven criteria that should be respected when designing error messages for a type inference system.

1. **Correct**. Both the *detection* and the *reporting* of an error should be correct. An error should only be detected when the formal specification of the language specifies that the program is in an erroneous state. This property is therefore quite strict. The soundness proof of the language should specify exactly when an error should occur. All reported errors should contribute in some way to the type conflict.

2. **Precise**. Each conflicting site should be located in the smallest useful amount of source text. The source text should therefore also be relevant. The relationship between the conflicting type and the site from which it was inferred should be simple and clear.

3. **Succinct**. The error report should maximize useful information and minimize non-useful information. There is a subtle balance between the two. Long and verbose explanations are tedious to read while short and terse explanations are hard to understand. Furthermore, assigning an error message to one of these specifications is open to interpretation. For a novice programmer, a longer, more explanatory, error may be useful whereas for an expert programmer, a shorter, more concise error is enough.

4. **A-mechanical**. An error report should not reproduce large amounts of counter-intuitive mechanical inference. In other words, the underlying inference process should not become visible to the user. GHC, the standard compiler for Haskell, often breaks this rule and shows error messages with *type variables* in them that do not occur in the program but were inferred during type inference.

5. **Source-based**. This rule goes hand in hand with rule 4. Only code from the source program should be used in the error message and no core syntax from any intermediate languages should be shown. This introduces a hard problem for embedded Domain-Specific Languages (DSLs) as the language that they are built on, becomes an intermediate language. However, the compiler does not know this and the errors are completely based on this base language.

6. **Unbiased**. The error should not depend on any particular order of inference of constraints. This rule is irrevocably violated by algorithms W and M which introduce a bias during type inference itself, as explained in section 3.2.1. Yang et al. use $f\ x = (x\ 1, x\ True)$ as an example where x may have two types. The error message should not point out $x\ 1$ or $x\ True$ as the mistake but should report both and let the programmer decide which to blame.

7. **Comprehensive**. Ideally, all sites that contribute to the error should be reported. This way, the programmer can reason about the error from the reported sites without having to look at other parts of the program to locate the exact location of the error.

Yang et al. continue by introducing two types of improved type error diagnosis systems: *Explanation systems* and *Error reporting systems* or *Reparation systems* as they are called by Heeren [16]. Furthermore, the literature introduces two other systems under the names of *Type error slicing* and *Interaction systems*.

3.3.2 Explanation systems

Explanation systems focus on providing a better explanation of how the *type inference* leads to a type error. There have been several error explanation systems.

Wand Wand [37] designed a system that keeps a recording of the pieces of code that contribute to the type deductions. This information is used to explain why the errors happen. The records are a combination of function applications and substitutions that cause the error. Each such application is reported. Where the inconsistency is found depends on the order of traversal of the syntax tree during analysis. Furthermore, the number of candidate error sites is also decided by the programming style.

Yang Yang [38] proposes a system called U_{AE} that eliminates the left-to-right bias of algorithms W and M and can point out all type conflicts automatically. Each sub-expression in an application is typed independently and results in an *assumption environment*. Such an assumption environment contains the local type constrains for every variable in the sub-expression. All assumption environments are unified at the top level of the Abstract Syntax Tree (AST), removing the bias. When U_{AE} fails, algorithm M is used over the remainder of the program to find the most fine-grained error site.

Yang, Michaelson, and Trinder Yang et al. [21] discusses the differences between how a type inference algorithm sees polymorphic types and how humans perceive them. The bias that W introduces is often not the way we humans would infer the type of a function. A human often focuses more on known type literals and uses these to infer the types and tends to avoid type variables at all. Yang et al. designed a new inference algorithm H that stores human-like explanations for the types that are inferred and mimics the identified human inference techniques. It is built on top of U_{AE}. The explanations that are generated are, according to the paper, both succinct and non-repetitive.

Yang et al. [39] argue that, based on their manifesto, explanation systems often fail on several key points.

- They are not **succinct**. Sometimes, the textual explanation has so much information, that it becomes rapidly tedious. The explanations are often found too detailed to be of real help.

- Explanations are **mechanical**. Because the systems try to explain the *inference* steps, type variables are often used in the explanation messages. To understand these, the programmer must have knowledge about the underlying inference process, which is often not the case.

- Not source-based. The explanations use text obtained from the AST, rather than the original program text.
- Biased. Most systems are based (at that time) on the W algorithm, which has a *left to right* bias.

3.3.3 Reparation systems

Reparation systems try to report the most appropriate location for a type inconsistency. *One* site is selected from a list of candidate locations because it is believed to be the location of the mistake [16].

Johnson and Walz Johnson et al. [19] introduce a maximum flow approach to determine the most likely cause of the type error. They recognize that error indications should be complete but parsimonious: the user should only see everything that contributed directly to the error but nothing more. The user should be drawn to the correct anomalies. The paper notices that in the case of an error, the variables in the error are used correctly most of the time and that only a small number of uses are conflicting. They thus want to draw the attention of the user towards those small numbers, where the error most likely resides.

McAdam McAdam [24] uses a graph to represent types of fragments of the program. Vertices in the graph can either be a program fragment or can represent a type. The paper shows that it is possible to extract assumption environments from these graphs, using the techniques from Bernstein et al. [2] and thus reducing bias in the type inference process. The graphs can be traversed to find an explanation for the type error.

Yang et al. [39] tested several repair systems against their manifesto. Most techniques were fairly mechanical, meaning that the underlying type inference system was visible in the error messages. What is interesting to note, is that most systems do not have a bias. Through the use of U_{AE} or other techniques, the systems can select a single location for the error, by considering all locations with equal probabilities. As a result, the bias often disappears.

3.3.4 Program slicing

In a type error slicing system, the type inconsistency is reported as a set of contributing program points. Such a set of points is called a *program slice* [10] and contains the location of the actual mistake. Other parts of the program thus do not need further inspection from the programmer. These systems do not use standard textual output but editors and other visualization tools to highlight parts of a program.

Schilling Schilling [28] uses a carefully chosen function to show how difficult it is to pinpoint the exact location of a type error in the source code. GHC often only reports one error where there may be more culprits. Multiple pieces of code can be adapted to solve the error. Program slicing means that all parts of the program that contribute to the error should be highlighted. Changes to these locations *may* remove the error. Schillings method does not use a type inference system in its back end. When this is the case, a *constrainted based type inference system* would be the best option because of the fact that constraints are easier to link to the source code.

Haack and Wells Haack et al. [10] recognize that algorithms W, M and U_{AE} often fail to identify the real location of a type error, albeit giving different locations. The algorithms identify one node of the AST that *participates* in the type error, but will often be the wrong node to *blame*. The presented algorithm is able to compute minimal type error slices by calculating minimal unsolvable constraint sets of equality type constraints. This constraint set is then used to highlight the linked source code. To this end, the algorithm introduces *labeled unification* which is similar to a basic unification algorithm apart from the fact that labels are added to the constraints to keep track of the location in the source program. These labels are similar to what is used for TOP and Rhodium.

Heeren [16] discusses that program slices give a truthful impression to quickly discover the actual problem in the program. It is however questionable whether the technique should be used as a stand-alone type error diagnosis system. A disadvantage of the technique is that types are no longer related to the expressions in the source code which means that the programmer should be completely aware of the types.

```
Ex1.hs> :debug palin
reverse :: [a] -> [[a]]
Ex1.hs> is this type correct> n
flip :: (a -> b -> c) -> b -> a -> c
Ex1.hs> is this type correct> y
foldl :: (a -> b -> a) -> a -> [b] -> [a]
Ex1.hs> is this type correct> n
type error - contributing locations
foldl f z [] = [z]
foldl f z (x:xs) = foldl f (f z x) xs
```

Figure 3.4: Example of a debugging session with Chameleon (taken from [16])

3.3.5 Interaction systems

Interaction systems aim to narrow down the location of a type error by interacting with the programmer. The advantage of the technique is that the system can become aware of the programmer's intentions and assumptions. An interactive system can help debug the system and guide the user to the location of the error.

A well-known interactive system is Chameleon, as designed by Stuckey et al. [35]. The system contains an interactive programming environment to debug type errors. It supports Haskell with some extensions and can be used as a front-end to existing Haskell compilers. The typing problem is mapped to a set of constraints. Each type constraint has the program code attached that is responsible for the restriction. Constraint handling rules (CHRs) are then used to manipulate the set of constraints at hand. This is done by either simplifying or propagating them. In case of an inconsistency, the minimal unsatisfiable set of constraints is determined, which are then used to underline parts of the program that contain an error. This works similarly to the approach Haack et al. used. Using this information, the system may request reasons why a certain type was chosen for a certain part of the program and thus obtain more information about the exact location of the error. Figure 3.4 shows an example debugging session in Chameleon.

3.3.6 Heuristics

As discussed in sections 3.2.4 and 3.2.5, both TOP and Rhodium use heuristics to increase error reporting quality. Both systems could be classified *reporting systems* that try to report the most relevant error. In this process, a *heuristic* is a function that can indicate the likelihood that a certain error has a specific cause. Knowing the cause greatly helps in the quality of the error report. Hage and Heeren [12] propose two types of heuristics: *filter heuristics* and *voting heuristics*. Both types of heuristics work over an error path that has been gathered after an inconsistency has been found in the constraint system.

Filter heuristics Filter heuristics are allowed to remove certain constraints from the error path. They are given a list of constraints that form the error path and determine which constraints are still able to be blamed. An example of a filter heuristic is the *Avoid unwanted constraints* heuristic. This heuristic uses information given to the constraint to remove constraints that should never be blamed. This is, for example, needed in case of a let binding. When an error occurs in one, the complete let binding should never be blamed. The cause of the error always occurs somewhere in the let body. Therefore, the constraint that represents the complete body should be removed.

Voting heuristics A voting heuristic is allowed to vote on the likeliness that a certain error is caused by a certain constraint. The heuristic is presented one constraint at a time and is allowed to attach a weight to it that represents how sure it is that the considered constraint is to blame. The function that performs this is called the *selector*. The *selector* is different for each defined voting heuristic. When the selector does not find the error for which it is designed, no weight is returned. All the voting heuristics compete against each other. The constraints with the highest weights are chosen to remain in the error path. All other constraints are removed.

3.4 Type error diagnosis for DSLs

Domain-Specific Languages or DSLs are languages that are restricted to a specific domain. Such a language could be embedded in other existing programming languages, such as Haskell, in which case we call it an embedded DSL. Embedding a DSL in an existing language brings forth the advantage of reusing the techniques that are already implemented. Some techniques that can be reused are code generation, optimization, and type checking.

Type checking an embedded DSL poses a problem, however. An embedded DSL is often coded as a library. When a type error occurs, the underlying implementation of the library in the mother language is revealed. The programmer often finds no support in such error messages as they are often complicated and form no relation with the actual program that the programmer is writing. In an ideal world, the type system displays an error designed for the DSL. In his Ph.D. thesis, Serrano describes this process in detail [32].

3.4.1 Type class directives

Hage and Heeren [13, 16] describe type class directives as a way to guide type errors in DSLs that use type classes. They propose four different directives that each allow the builder of the DSL to specify how type classes defined in the DSL may be used. First, they propose the *never directive* which indicates that a certain type class instance can never occur. An example would be $Eq\ (a \to b)$ as it is impossible to check whether two functions are equal. The next directive they propose is the *close directive*, which specifies that all instances of this type class are known and no other may be specified. Some defined type classes in a DSL are meant to be untouched and with this directive the user receives a specialized error that no instance can be created for such a type class. The *disjoint directive* states that a type can never be an instance of multiple classes at once. This directive always specifies pairs of type classes. One example of a pair of type classes that may be assigned this directive is *Fractional* and *Integral* because a number can be Fractional or Integral but not both. Finally, they specify the *defaulting directive* which can be used to guide the type inference system in case of ambiguity. Let us take the following example program: $main = show\ []$. the function *show* is part of the type class *Show* but how do we know, based on the information at hand, what instance of *Show* to use? We are not, and this is where the defaulting directive helps us. It allows us to specify the defaults in case the type inferencer runs into an ambiguous case.

3.4.2 Specialized error messages

The old Helium compiler allows programmers of DSLs to specify *specialized type rules* that are able to induce *specialized error messages* tinkered to make sense towards the DSL. Such type rules can be provided externally and no tinkering with the type inferencer itself is needed. Specialized type rules are specified in a *.type* file and have been designed by Hage and Heeren [14]. Let us consider the following example as taken from their paper:

```
  x :: t1;        y :: t2;
----------------------------
   x <$> y :: Parser s b;

t1 == a1 -> b : left operand is not a function
t2 == Parser s a2 : right operand is not a parser
a1 == a2 : function cannot be applied to parser's result
```

This type rule is written for the `<$>` operator for parser combinators. The rule starts with defining the argument and result types of the function inside a deduction rule. Based on the type variables defined in this rule, the programmer is then able to define constraints together with error messages that should be shown when they are violated. For example, it is expected that `t1` should be a function type and that `t2` should be of type `Parser s a2`. During the definition of constraints, it is also possible to define new type variables, like `a1` and `a2`, and to use them in new rules. The rule `a1 == a2` specifies that the result type of the parser in `y` must be equal to the argument that the function `x` defines.

Phasing Through *phasing*, the exact order of the type inference process can be guided. As a result, the programmer can design the type inference process in an intuitive way. Our previous example now would look as follows [14].

```
 x :: t1;   y :: t2;
---------------------
   x <$> y :: t3;

phase 6
t2 == Parser s1 a2 : right operand is not a parser
t3 == Parser s2 b2 : result type is not a parser
phase 7
s1 == s2 : parser has an incorrect symbol type
phase 8
t1 == a1 -> b1 : left operand is not a function
a1 == a2 : function cannot be applied to parser's result
b1 == b2 : parser has an incorrect result type
```

Phases 1 to 5 are default phases of the type inferencer that will be executed before the specialized type rule. Notice that that inferencer will now execute phases 6 to 8 after one another. As a result, the type inferencer will first check whether `t2` and `t3` are of type *Parser* (phase 6). Next, it will check whether the symbol types are equal (phase 7), and last but not least, it will check whether the flow of argument and result types are correct (phase 8).

Siblings Last but not least, a specialized type rule can specify *sibling functions.* A sibling function is a function that is very similar to the function for which the type rule is written. Often, such functions may be erroneously switched up by the programmer. For example, (:) and ++ are often switched up. When told that these functions are siblings, the type inferencer may try to switch out one function for the other and check if that solves the type error.

Chapter 4

Implementation of Type Families in GHC

This section describes how the static checks and type inference are designed for type families. Before we can improve type error diagnosis for type families with the toolset that Helium provides, we need to implement Type Families as a language construct. The design is heavily based on the GHC compiler for Haskell as Helium is meant to mimic GHC as closely as possible. This section first shortly describes the use of type families in the type inference process. It then proceeds to discuss general static checks that the compiler will have to perform. We then consider open, closed, and injective type families separately and discuss their general usage in the type inference process and important static checks that ensure their correct behavior and termination. Finally, this section discusses how the X in Rhodium should be extended to incorporate the type inference rules for type families.

4.1 Type families in type inference

During type inference, type family instances are considered *top-level axioms* that represent a left-to-right rewrite rule to simplify a type family application to its definition. Which instance is chosen depends on the arguments given to the family. How a definition is chosen depends on the type of type family that is being used. We discuss the implementations in their respective sections. A simple example should illustrate the simplification of types using type families.

```
type family F a
type instance F Int = Float
type instance F Float = Int

f :: (F Int) -> Float
```

The type of function f will be simplified by type family F through its first instance $F\ Int$. The new type of f becomes `Float -> Float` as per the definition of the type instance.

4.2 General static checks

The static checks discussed in this section should hold for every flavor of type family. As an associated type synonym may only appear open, section 4.2.3 naturally does not hold for closed type families.

4.2.1 Unique names

The first static check that must be implemented, is one that already exists for other top-level constructs, like data definitions and type classes: there may only be one declaration of a certain name. This is not only true among type families alone. This rule should apply across every type of top-level construct. Therefore, if a name is already used for a type family, it cannot be used to declare a new data type or type class and vice versa.

4.2.2 Arities of instances

The arity of a type family is the number of arguments with which it is declared. For example,

```
type family F a b
```

has an arity of two. Every instance that is made for this type family should fully saturate this arity. Therefore, all type instances for F, should be defined with two arguments.

```
type instance F Int Bool = Float -- Correct!
type instance F Bool = Bool -- Incorrect, missing one argument!
type instance F Int Bool String = Int -- Incorrect, too many arguments!
```

The first type instance definition is a correct one, the arity is fully saturated. The second one misses an argument and is thus incorrect. Naturally, too many arguments are also incorrect, as the third case shows.

4.2.3 Arguments of associated type synonyms

According to Chakravarty et al [5], the arguments of an associated type synonym must match the instance head type parameters that they are linked to. This is best explained using an example, as taken from their paper.

```
class C a where
    type S a k

instance (C a) => C [a] where
    type S [a] k = (a, k a)
```

Here, the class declaration specifies that the type parameter a is used in both the class head and the associated type synonym. Every instance made henceforth should resurrect this declaration. The instance instantiates a to $[a]$ and therefore the first argument of S must also become $[a]$. Notice that k is not present in the instance head and is thus not subject to this restriction.

4.3 Open type families

As mentioned in section 2.3, type family instances for open type families may be declared anywhere as long as the original declaration is known. Associated type synonyms and open type families have the exact same behavior except for the static check discussed in 4.2.3.

4.3.1 Instance application

Applying an instance of an open type family to a family type is quite straightforward. The simplification process checks whether the given type *unifies* with a type family instance based on its arguments. If this is the case, the family type is substituted by the instance definition on the right-hand side of the instance and the inference process continues. If not, then a new instance is tried. When no instances are able to simplify the type, the inferencer may potentially produce an error as further type inference may fail.

4.3.2 Static checks

The static checks we discuss in this section are restrictions imposed to guarantee a *decidable confluence* for the top-level instances of type families. The checks guarantee that the type inferencer will always be able to deterministically determine which instance can be used to simplify a given type. Furthermore, we discuss checks that guarantee termination of the type inference process. As a result, the type inferencer always finishes in finite time.

Confluence: Overlapping instances Open type family declarations and their instances can be declared anywhere across different modules, given that the declaration is compiled before any of its instances. As a result, we cannot assume anything about the order of occurrence. This affects the freedom of defining type instances. Schrijvers et al [29] impose the restriction that *instance heads may not overlap*. This restriction ensures that, during type inference, only one type instance can match, and thus *confluence* is decidable. Consider the following type instances and function.

```
type family G a
type instance G Int = Float
type instance G Int = Bool

f :: (G Int) -> Bool
```

To be able to correctly type the function f, `G Int` must be converted to a simpler type. However, two type definitions for `G Int` exist. It becomes impossible for the type inference system to know which one to choose, which violates the *confluence* rule. To ensure type system correctness, choosing the axiom must be decidable and therefore type instance heads may not *overlap*. In case we have two type family instances $F\ \bar{\sigma}_1 \sim \tau_1$ and $F\ \bar{\sigma}_2 \sim \tau_2$ where $\bar{\sigma}$ represent the type arguments (lhs) and τ represents the type family definition (rhs), we define these instances as non-overlapping if

1. $\exists i : \neg(\sigma_{1,i} \sim \sigma_{2,i})$. There exists a pair of type arguments between the two instances that does not unify,
2. or $\forall i : (\sigma_{1,i} \sim \sigma_{2,i}) \wedge \theta(\tau_1) = \theta(\tau_2)$. All arguments unify with each other and both result types are equal after applying the resulting substitution.

In all other cases, two instances are overlapping. Let us clarify this with an example.

```
type instance F (a, Int) = [a]
type instance F (Int, b) = [b]    -- overlap permitted

type instance G (a, Int)  = [a]
type instance G (Char, a) = [a]   -- ILLEGAL overlap, as [Char] /= [Int]
```

For type family `F`, `a` can be substituted with `Int` and `b` can be substituted with `Int`. We thus end up in case 2. We end up with the substitution θ with $a \mapsto Int$ and $b \mapsto Int$. Applying this substitution yields us `[Int]` and `[Int]` for both result types respectively. They are equal and the overlap is thus permitted. For `G`, we get a substitution θ with $a \mapsto Char$ and $a \mapsto Int$ which, after applying it, gives us `[Char]` and `[Int]` respectively. This violates the rule and therefore is illegal.

Confluence: No type families in type family arguments This rule is analogous to normal, value level, functions where it is required that the patterns use only variables or constructors, but not functions. Therefore, the following example is illegal.

```
type family H a
type instance H (F a) = a -- Illegal, parameter has a type family
```

The intuition here is that it becomes difficult for the compiler to determine whether two type families overlap. `F a` could be simplified using many instances, depending on the instance of a. Furthermore, `F a` could simplify to a simple type for which another instance is defined. As a result, we end up in a situation in which confluence is endangered.

Termination The main principle for establishing termination of the type inference process, is guaranteeing decreasing calls. Such calls guarantee that in each type inference step, the process gets closer to a terminating state. Schrijvers et al [30] do not propose a state-of-the-art termination check but instead pose three restrictions upon type family instances. For the following given type instance

```
type instance F t1 .. tn = t
```

we require that for each type family application `(G s1 ..  sm)` in `t`, that

1. `s1 ..  sm` do not contain any type family constructors,
2. the total number of symbols (data type constructors and type variables) in `s1 ..  sm` is strictly smaller than in `t1 ..  tn`, and
3. for every type variable `a`, `a` occurs in `s1 ..  sm` at most as often as in `t1 ..  tn`.

The first rule requires that type families may not appear nested in the function body `t`. The reason for this rule is to prevent increasing type families because of the recursive definition. In the example

```haskell
type instance F Int Int = G (F Int Int)
```

the type inference process may not terminate. A type family `F Int Int` is replaced by `G (F Int Int)`, after which we may apply the same instance again, continuously enlarging the type family. The second and the third rule make sure that the function body of the type family always becomes smaller and that no extra, possibly non-terminating constructs are introduced. These restrictions are quite harsh and violate the completeness property of type families: not every instance that in theory would work, is accepted by the compiler. For example, a constraint like $a \sim [F\ a]$ could not occur because of above rules [30].

4.4 Closed type families

Closed type family instances may only be declared in the where clause of the original declaration. This opens it up for the possibility of letting instances overlap. The instances declared in the where clause are considered from top to bottom by the type inferencer when it tries to find a matching instance for a type.

4.4.1 Instance application

The main difference between open and closed type families is how instances are applied to types. As described above, instances are considered from top to bottom. The introduction of this order allows declared instances to overlap to a certain degree. Formally, as described by Eisenberg et al. [8], an instance to use in a given type family application (the target) will be selected if and only if the following two conditions hold:

1. There is a substitution from the variables in the *lhs* of the instance that makes the left-hand side of the branch coincide with the target. Therefore, the instance must unify with the target. We say that the target and the instance *match*.

2. For each previous instance (the ones defined above it), either the *lhs* of that instance is *apart* from the considered instance, *or* the instance is *compatible* with the considered instance.

Where the first case makes immediate sense, the second introduces two new terms: *apart* and *compatible*. We define and discuss them.

1. ***Apart***. Two types are *apart* when one cannot simplify to the other, even after arbitrary type-family simplifications. To provide a formal definition, we first need to introduce *flattening* of a type as defined by [8, 36]. To *flatten* a type τ into τ', process τ in a top-down fashion, replacing every type family application that is encountered, with a type variable. Two or more syntactically identical type family applications are flattened to the same variable; distinct type family applications are flattened to distinct fresh variables. With the notion of flattening, we can now formally define *apartness*. For an instance argument type ρ and a target argument type τ, $\text{apart}(\rho, \tau)$ is defined as follows. Let $\tau' = \text{flatten}(\tau)$ and check $\text{unify}(\rho, \tau')$. If this unification fails, then ρ and τ are apart. More succinctly: $\text{apart}(\rho, \tau) = \neg\text{unify}(\rho, \text{flatten}(\tau))$. The complete apartness check, applies *apart* to all arguments of the type family application and instance: $\text{apart}(lhs_p, \bar{\tau})$.

2. ***Compatible***. The compatibility check is similar to how overlapping is checked for open type families (section 4.3.2). Formally, two instances p and q are compatible iff $\Omega_1(lhs_p) = \Omega_2(lhs_q)$ implies $\Omega_1(rhs_p) = \Omega_2(rhs_q)$. To check if this holds, we introduce the test $\text{compat}(p, q)$ that checks that $\text{unify}(lhs_p, lhs_q) = \Omega$ implies that $\Omega(rhs_p) = \Omega(rhs_q)$. If $\text{unify}(lhs_p, lhs_q)$ fails, $\text{compat}(p, q)$ holds vacuously. Notice that this definition is equal to the definition of overlap in section 4.3.2.

The definitions of *apart* and *compatible* are crucial to allow for overlap between the defined type family instances. The apartness check ensures us that there is no way that a previous instance may be used to simplify the target. Even though it may have been the case that the instance did not match. There are situations in which, after further simplification of the constraint system, the previous instance will match. Eisenberg et al. [8] show this with the following example.

```haskell
type family Equal a b :: Bool where
    Equal a a = True
    Equal b c = False
```

```haskell
type family FunIf (b :: Bool) :: * where
    FunIf True = Int -> Int
    FunIf False = ()

bad :: d -> FunIf (Equal Bool d)
bad = ()
segFault :: Int
segFault = bad True 5
```

Without the apartness check, the following may happen. If we simplify `Equal Bool d` to `False`, we can show that `bad` is well typed since `FunIf False` is `()`. But then `segFault` calls `bad` with `d` instantiated to `True` which results in `FunIf (Equal Bool Bool)`. This reduces to `Int -> Int` and as a result, `bad` does not type check. We wrongly reduced `FunIf (Equal Bool d)` to `False` because it depends on how `d` is instantiated. The substitution for `d` may thus allow it to be reduced with the first type family instance. In this case, the instances are thus not apart and the second rule should not be applied yet.

However, when both instances would reduce to the same type family definition no matter to what type a type variable is instantiated, then we should ignore the apartness check. This occurs when two instances are found to be *compatible*.

4.4.2 Static checks

From the static checks discussed for open type families (section 4.3.2), all checks also hold for closed type families, except for the one that checks for overlapping. In this section, we discuss one more static check for closed type families, an optimization for the compatibility check, and a generalization for open type families.

No top-level instances for closed families As already discussed, closed type families may not introduce stand-alone instances. This should be checked before type inference.

Optimization: pre-compute compatibility There are cases in which the apartness check is not needed during instance application. This is the case when two type family instances are *not compatible*. According to the definitions of apart and compatibility, non-compatibility implies apartness. Thus, when two instances are not compatible, we do not have to check their apartness.

Compatibility for open type families As mentioned, the compatibility check for closed type families is equal to the static check for overlapping open type families. We generalize the overlapping check for open type families thus as follows: two open type family instances p and q are overlapping when $\text{compat}(p, q)$.

4.5 Injective type families

An injective type family is a type family that has injective properties. Such properties allow the type inference system to become more expressive, resulting in more programs being type correct. This section first discusses what injectivity is and why it is not normally inferred for type families. We discuss complications that may arise during type checking and inferring and how these are remedied using static checks. Last but not least, we discuss what injectivity enables for type checking and inference.

4.5.1 Injectivity of type families

Stolarek et al. [34] formally define the injectivity of a type family as follows.

> A type family F is n-injective (i.e. injective in its n'th argument) iff $\forall \bar{\sigma}, \bar{\tau} : F\ \bar{\sigma} \sim F\ \bar{\tau} \implies \sigma_n \sim \tau_n$.

Here, $\sigma \sim \tau$ means that we have a proof of equality of types σ and τ. The definition thus tells us that if we have a proof that $F\ \bar{\sigma}$ is equal to $F\ \bar{\tau}$, then we have a proof that σ_n and τ_n are equal. Moreover, if we know that $F\ \bar{\tau} \sim \tau'$ and τ' is known, we can discover values of the injective arguments $\bar{\tau}_n$ by looking at the defining equation (the instance) of F that has rhs matching τ'. The annotation for injective type families is discussed in section 2.5.

Why not infer injectivity? Injectivity is not automatically inferred for both open and closed type families. For open type families, inferring it is generally impossible. Equations may spread across modules and can be added at any time. Inferring injectivity would be based only on those equations in the declaring module which would lead to unexpected behavior as the next module may ruin the previously found injectivity by introducing an instance that violates the above definition. Stolarek et al. [34] argue that for closed type families, injectivity may be inferred but that it would be the wrong design decision as it could lead to unexpected behavior during code refactoring. When injectivity is annotated, it is clear to the programmer that the property is required for the program to compile. When it is inferred, the programmer may be unaware that the program relies on an inferred-injective type family. The user may add a new equation that breaks the injective property which then results in a large set of errors. Such errors could be very distant and may even be in a different package. One can see that this would greatly confuse the user. Therefore, type families may only be injective when they are declared to be so.

Verifying injectivity annotations Stolarek et al. [34] show that checking injectivity is more subtle than it may initially appear. They present three 'awkward cases'.

1. ***Injectivity is not Compositional***. This case is best shown using the example from their paper.

   ```
   type family F1 a = r | r -> a
   type instance F1 [a] = G a
   type instance F1 (Maybe a) = H a

   type instance G Int = Bool
   type instance H Bool = Bool
   ```

   Sadly, there is no guarantee that `F1` is injective as it is claimed to be by its annotation. Even when `G` and `H` are. In our example, `G` and `H` are naturally injective, but `F1` is not. We could, for example, have the following type equality:

   $$F1\ [Int] \sim G\ Int \sim Bool \sim H\ Bool \sim F1\ (Maybe\ Bool)$$

   which shows that F1 is not injective and that injectivity is thus not a compositional property. A composition of injective type families does not necessarily make the newly declared type family injective.

2. ***The Right Hand Side cannot be a Bare Variable or Type Family*** Again, we illustrate this case by using an example:

   ```
   type family W1 a = r | r -> a
   type instance W1 [a] = a
   ```

   This type family does not satisfy the definition for injectivity of type families (section 4.5.1). We clearly have a proof of `W1 [W1 Int] ∼ W1 Int` as defined by the instance. However, if `W1` follows the definition of injectivity, then we would also have a proof of `[W1 Int] ∼ Int`, which is plainly false. Similarly, we can write a type family with an instance with a type family in its rhs.

   ```
   type family W2 a = r | r -> a
   type family W2 [a] = W2 a
   ```

   For this family, we have a proof for `W2 [Int] ∼ W2 Int` which, according to the definition of injectivity, would also give us a proof for `[Int] ∼ Int` which is also plainly false.

3. ***Infinite types***. Again the paper [34] describes this case with an example.

   ```
   type family Z a = r | r -> a
   type instance Z [a] = (a,a)
   type instance Z (Maybe b) = (b, [b])
   ```

At first glance, one might expect the injectivity annotation to hold. `(a,a)` and `(b, [b])` look like they may never unify and cause any problems. However, type families themselves destroy this idea. Let us consider the following type family.

```
type family G a
type instance G a = [G a]
```

We now choose `a = b = G Int` for type family `Z`. As a result, we obtain the following type equality:

$$Z\ [G\ Int] \sim (G\ Int, G\ Int) \sim (G\ Int, [G\ Int]) \sim Z\ (Maybe\ (G\ Int))$$

Applying our definition of injectivity on the first and last of this inequality gives us $[G\ Int] \sim Maybe\ (G\ Int)$, which is definitely unsound. GHC's default behavior is that type family definitions are restricted in the sense that the body type must be smaller than the type family application to ensure termination. The static checks that ensure this are discussed in section 4.3.2.

4.5.2 Static checks

The static checks surrounding injective type families are tasked with verifying the injectivity annotation given by the programmer and handling the discussed awkward cases. Stolarek et al. define the *injectivity check* as follows:

1. For every instance $F\ \bar{\sigma} = \tau$:

   a τ is not a type family application, and

   b if $\tau = a_i$ (for some type variable a_i), then $\bar{\sigma} = \bar{a}$.

2. Every pair of equations $F\ \bar{\sigma}_i = \tau_i$ and $F\ \bar{\sigma}_j = \tau_j$ (including $i = j$) is *pairwise-n-injective*.

Clause 1 deals with awkward case 2, by rejecting instances whose *rhs* is a bare type variable or function call. In practice, this restriction is barely noticeable as any instance rejected by clause 1 would also be rejected by clause 2 if there is more than one instance.

Clause 2 compares the instances of a type family pairwise. The intuition of the check is that two equations are pairwise-n-injective if, when the *rhs*'s are the same, then the n'th argument on the left-hand side is also the same. The sameness check over the *rhs*'s here naturally means that they need to be unifiable. However, classic unification is not enough in this situation. Consider an example where the type instance *rhs* contains a type family application.

```
type family G1 a = r | r -> a
type instance G1 [a] = [G a]
type instance G1 Int = [Bool]
```

Here we assume that `G` is another type family known to be injective. The *rhs*'s do not unify under classical unification because of the lack of a unifying substitution. Therefore, such an algorithm would accept `G1` as injective. This would however be wrong: we might have an instance for `G`, `G Int = Bool`, in which case `G1` is not injective. To fix this, we need a variant of the unification algorithm that *treats a type family application as potentially unifiable with any other type.* Stolarek et al. [34] define a *pre-unification* algorithm $U(\sigma, \tau)\ \theta$ as defined in figure 4.1.

The algorithm takes two types, a and τ, and a substitution θ and returns either **Nothing** or **Just** ϕ, where ϕ extends θ. The definition is similar to that of classical unification except:

- Equations (8) and (9) deal with the case of type function application: it immediately succeeds without extending the substitution.
- Equation (7) allows U to recurse into the injective arguments of a type-function application.
- Equation (2) would fail in classical unification (the occurs check). In U, it succeeds immediately, but without extending the substitution. This has to do with awkward case 3, which deals with infinite types.

Last but not least, there is a small subtlety when checking closed type families. Consider the following example:

$$
\begin{array}{rcll}
U(a,\tau)\ \theta & = & U(\theta(a),\tau)\ \theta & a \in \mathsf{dom}(\theta) \quad (1)\\
U(a,\tau)\ \theta & = & \mathbf{Just}\ \theta & a \in \mathit{ftv}(\theta(\tau)) \quad (2)\\
U(a,\tau)\ \theta & = & \mathbf{Just}\ ([a \mapsto \theta(\tau)] \circ \theta) & a \notin \mathit{ftv}(\theta(\tau)) \quad (3)\\
U(\tau,a)\ \theta & = & U(a,\tau)\ \theta & \quad (4)\\
U(\sigma_1\,\sigma_2,\tau_1\,\tau_2)\ \theta & = & U(\sigma_1,\tau_1)\ \theta \ggg= U(\sigma_2,\tau_2) & \quad (5)\\
U(H,H)\ \theta & = & \mathbf{Just}\ \theta & \quad (6)\\
U(F\ \overline{\sigma},F\ \overline{\tau})\ \theta & = & U(\sigma_i,\tau_i)\ \theta \ggg= & F \text{ is } i\text{-injective} \quad (7)\\
 & & \ldots \qquad \ggg= & \text{...etc...}\\
 & & U(\sigma_j,\tau_j) & F \text{ is } j\text{-injective}\\
U(F\ \overline{\sigma},\tau)\ \theta & = & \mathbf{Just}\ \theta & \quad (8)\\
U(\tau,F\ \overline{\sigma})\ \theta & = & \mathbf{Just}\ \theta & \quad (9)\\
U(\sigma,\tau)\ \theta & = & \mathbf{Nothing} & \quad (10)
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{Just}\ \theta \ggg= k & = & k\ \theta\\
\mathbf{Nothing} \ggg= k & = & \mathbf{Nothing}
\end{array}
$$

Figure 4.1: Pre-unification algorithm U (taken from [34]). The algorithm takes two types, a and τ and performs pre-unification on them. The algorithm returns a substitution θ if it succeeds and nothing if it fails. It is similar to classic unification except for the cases discussed in section 4.5.2.

```
type family G2 a = r | r -> a where
    G2 Int = Bool
    G2 Bool = Int
    G2 a = a
```

`G2` is injective. However, when we compare the rhs's of the first and the third case, we obtain the substitution $[a \mapsto \mathit{Bool}]$. Applying this to the lhs of the third case results in the arguments not being equal. However, the third equation cannot reduce with $[a \mapsto \mathit{Bool}]$ because it is overshadowed by the second case. This condition is easy to check for. If after substituting the lhs's, they are different, we know that one of them cannot fire under that substitution, and thus there is no problem with injectivity.

We can now provide the complete definition for the pairwise-injectivity check. A pair of equations $F\ \bar{\sigma}_i = \tau_i$ and $F\ \bar{\sigma}_j = \tau_j$, whose variables are *disjoint*, are pairwise-n-injective iff either

1. $U(\tau_i, \tau_j) = \mathbf{Nothing}$, or
2. $U(\tau_i, \tau_j) = \mathbf{Just}\ \phi$, and
   - a $\theta(\sigma_{i\ n}) = \theta(\sigma_{j\ n})$, or
   - b $F\ \theta(\bar{\sigma}_i)$ cannot reduce via equation i, or
   - c $F\ \theta(\bar{\sigma}_j)$ cannot reduce via equation j

cases b and c solve the subtlety involving closed type families. In case 1, the rhs's do not unify. In case two, they do and we need to check that both lhs's are equal for the n'th argument.

4.5.3 Type inference

Proving that an injectively defined type family is indeed injective has several benefits for the type inference process. First of all, it allows us to reduce the constraint $F\ a \sim F\ b$ to $a \sim b$ according to the original definition of injectivity. More importantly, the inference system is now able to do this *without* having to guess that $F\ a$ and $F\ b$ reduce to the same thing. Knowing $a \sim b$ may make some extra unifications apparent, making the type family more expressive.

Secondly, suppose we have the following type family and instance:

```
type family F a = r | r -> a
type instance F [a] = a -> a
```

and we are trying to solve a wanted constraint $F\ \alpha \sim (\mathit{Int} \to \mathit{Int})$, where α is a type variable in the constraint solving process. The defined type instance would match with the type instance defined for F and the instance becomes `F [Int] = Int -> Int`. Using this information and the fact that F is injective, allows us to deduce the constraint $\alpha \sim [\mathit{Int}]$ which again may make more unifications possible during the constraint solving process. Notice that instead of matching on the lhs of F, we now match on the rhs because the injectivity property tells us that every rhs has a unique lhs.

Chapter 5

Type Error Diagnosis for Type Families in GHC

This section describes the type error diagnosis quality for type families in the GHC Haskell compiler. We focus on this compiler because it is the official compiler for Haskell and Helium aims to closely mimic the Haskell standard implemented in it. The checks explained in section 4 take their toll on how intuitive type families are in Haskell. Many cases that should generally be possible are not, to guarantee the termination of the type inferencer and to keep the process decidable.

5.1 The difficulties

Overall, the difficult part in introducing type families for type error diagnosis, is that it becomes hard to find the *original type* that has to do with the error. After a type family application has been applied, the original type is gone and replaced by its definition. Furthermore, *injectivity* is often a difficult concept to grasp for even the more advanced functional programmers. The reason why injectivity properties are violated or why a non-injective type family is used as if they are injective is often hard to understand. Last but not least, type families are considered as *total* functions in GHC. A type family application is always accepted, even if there is no instance defined that can reduce it. When a type family is not reducible, it is left residual. This section considers several example programs accompanied by the error that is produced by GHC. We discuss for each its quality and how it could be improved.

5.2 Errors for static checks

Overall, errors considering the static checks for type families are quite clear in GHC. It must be said, however, that these checks do not impose the difficulties that the type inference process imposes. Let us first consider an example in which open type family instances conflict:

```haskell
type family Element container
type instance Element Int = Bool
type instance Element Int = Bool
```

GHC returns the following error:

```
main.hs:177:15: error:
    Conflicting family instance declarations:
      Element Int = Bool -- Defined at main.hs:177:15
      Element Int = Float -- Defined at main.hs:178:15
    |
177 | type instance Element Int = Bool
    |               ^^^^^^^
```

The error mentions exactly where the error resides and which instance declarations are at fault. The error blames line 177 specifically, however, which should not be needed. It may not be the wrongly implemented instance. That should be for the programmer to decide. When we consider an overlapping closed type family, the error converts into a warning:

```
type family H a b where
  H Int Int = Float
  H Int Int = Int

main.hs:206:3: warning:
    Type family instance equation is overlapped:
      H Int Int = Int -- Defined at main.hs:206:3
    |
206 |   H Int Int = Int
    |   ^^^^^^^^^^^^^^^
```

Because closed type family instances are considered from top to bottom, there is no risk for non-determinism during the type inference process. The only thing that may happen is that the program does not type check due to choosing the wrong instance.

A simple not saturated instance also behaves as it should and returns an error:

```
type family H a b where
  H Int Int = Float
  H Int = Int

main.hs:206:3: error:
    • Number of parameters must match family declaration; expected 2
    • In the type family declaration for ‘H’
    |
206 |   H Int = Int
    |   ^^^^^^^^^^^
```

The location is provided correctly and the error notifies the programmer exactly of what is wrong. However, when type variables are introduced, the error in GHC becomes confusing:

```
type family Append xs ys where
  Append '[] ys = ys
  Append (x:xs) = x : Append xs ys

main.hs:37:33: error:
    Not in scope: type variable ‘ys’
    Perhaps you meant ‘xs’ (line 37)
   |
37 |   Append (x:xs) = x : Append xs ys
   |
```

The source of the error is that in the second case *ys* as an argument is missing. A static check should catch this error by stating that the second case is not fully saturated. Apparently, a scope check for type variables is conducted before this and throws this error before the saturation check is even applied. This results in a confusing error from which the programmer now must deduce what the real problem is.

The errors related to the injectivity checks are well defined:

```
type family Y x = r | r -> x
type instance Y Int = Float
type instance Y Float = Int
type instance Y Bool = Int

main.hs:183:15: error:
    Type family equation right-hand sides overlap; this violates
    the family's injectivity annotation:
      Y Bool = Int -- Defined at main.hs:183:15
      Y Float = Int -- Defined at main.hs:182:15
    |
183 | type instance Y Bool = Int
    |
```

Not only does the error provide the programmer with the information of which combination of instances is erroneous, but it also provides a clear explanation. The right-hand sides overlap!

We leave out the rest of the static checks as they are quite trivial and GHC handles them well in terms of error diagnosis. Furthermore, the error diagnosis during type inference is much more interesting and harder to do, as we have seen in this document. We end this section with the notion that GHC apparently expects every error to provide the line on which the error occurs. In case the error results from two instances that conflict, this becomes confusing. It looks like GHC tells the programmer to blame exactly that line while in practice, both instances may be to blame. Which one to fix should be up to the programmer.

5.3 Errors during type inference

This section describes and discusses examples considering the difficulties mentioned in section 5.1. We consider type errors that result from nested type families. Furthermore, we look at the situation in which a type family is used injectively but is not annotated as such. We also consider situations in which the type inference system assumes the type family to be total but no usable instances exist.

5.3.1 No injectivity annotation

When a type family is annotated with an injectivity annotation, the compiler knows that this condition should hold and checks it. The errors returned when this check fails, are good. This, however, becomes another story when it is the other way around. If a type family that is not declared with an injectivity annotation, is used in an injective way, the compiler forgets that there is the possibility that it could be injective after all. We show this using the following example:

```haskell
type family Foo a
type instance Foo Char = Bool
type instance Foo Int = Double
type instance Foo Float = Int

class Bar t where
  clsF :: Foo t
instance Bar Char where
  clsF = True

main :: IO ()
main = print (clsF :: Bool)
```

Type family `Foo` complies with the injective property `r -> a` but is not annotated as such. The function `main` wants to print `clsF` which is explicitely typed as a bool. Because we know `Foo` is injective, we see that `Foo t ∼ Bool` should substitute `t` with `Char` because we can read the type family from right to left. However, as we discussed in section 4.5.1, the type inference system does not infer injectivity. This is not a problem if the error notifies the programmer correctly of the situation. GHC does not do this and provides the following error instead:

```
main.hs:117:3: error:
    • Couldn't match expected type: Foo t
                  with actual type: Foo t0
      NB: ‘Foo’ is a non-injective type family
      The type variable ‘t0’ is ambiguous
    • In the ambiguity check for ‘clsF’
      To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
      When checking the class method: clsF :: forall t. Bar t => Foo t
      In the class declaration for ‘Bar’
    |
117 |   clsF :: Foo t
    |   ^^^^^^^^^^^^^
```

A good thing is that the error mentions that the type family `Foo` is not injective. This could, however, mean two things: the type family does not have injective properties, which it obviously has, or the type family is not annotated as an injective type family. The error means the second situation but nonetheless, this may confuse the programmer. Furthermore, the error introduces the type variable `t0` as an *ambiguous* type variable. `t0` is not part of the source program which, according to section 3.3.1, is a bad thing. This `t0` was introduced in the ambiguity check that GHC performs on type signatures. In the ambiguity check, the constraint `Foo t ∼ Foo t0` is created and type-checked. Because `Foo` is not injective, this fails as we are not able to create the constraint `t ∼ t0` using the injectivity property. The second bullet point is where the confusion really starts. The compiler suggests turning on `AllowAmbiguousTypes` to fix the error, which is a language extension that allows more freedom with respect to ambiguous types. In this case, however, the error is not solved. Turning on the extension results in the following error:

```
main.hs:110:15: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘Foo t0’
      The type variable ‘t0’ is ambiguous
    • In the first argument of ‘print’, namely ‘(clsF :: Bool)’
      In the expression: print (clsF :: Bool)
      In an equation for ‘main’: main = print (clsF :: Bool)
    |
110 | main = print (clsF :: Bool)
    |               ^^^^
```

The ambiguity problem remains. The site blamed, however, is now different. Instead of the class declaration, the error now blames `clsF` in `main`. The inference system is not able to solve the constraint `Foo t ∼ Bool`. While this is certainly true and closer to the true cause of the error, we are still not quite there yet. The true source of the error, lacking an injectivity annotation, is now completely missing. A better error message would mention that `Foo` misses the injectivity annotation that enables the type inference system to solve this problem. The message could also provide a hint containing injectivity annotations that could work.

It could also be the case that an injectivity annotation was given, but that it is not expressive enough to solve the constraint. Then, we could provide a similar error, hinting towards the correct annotation.

5.3.2 Type families as total functions

Morris et al. [25] describe how type families, as implemented in Haskell, are always assumed total. Type families are open and extensible, so assuming they are total should be counterintuitive. Instead, one might see type families as partial to the extent of the type instances written for them. As a result of this totality, GHC sees a type family application as a type itself. This results in some interesting cases which we will introduce using an example:

```haskell
type family Y x
type instance Y Int = Float
-- type instance Y Float = Int
-- type instance Y Bool = Int

test :: Y Float -> Y Float
test x = x

test2 :: Y Float -> Int
test2 x = x

test3 :: Y Float -> Y Bool
test3 x = x
```

Type family `Y` only has one instance defined: `Y Int`. In a partial sense, this would mean that `Y` could only be correctly used with `Int` as its argument. Sadly, this is not true: the function `test` type checks! `Y Float` is seen as a type and as a result, we obtain the constraint `Y Float ∼ Y Float` which reduces to ϵ and thus poses no problems in the type inference system. The function `test2` does not type check and throws the following error:

```
main.hs:188:11: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Y Float’
    • In the expression: x
      In an equation for ‘test2’: test2 x = x
    |
188 | test2 x = x
    |           ^
```

The error tells us that `Int` does not match with `Y Float`, which is correct, the types are not the same. It however fails to mention the cause of the error, which is important: the programmer may have forgotten to create the instance for `Y Float`.

GHC also becomes confused when type checking function `test3`. It gives the following error:

```
main.hs:191:11: error:
    • Couldn't match expected type: Y Bool
                  with actual type: Y Float
      NB: ‘Y’ is a non-injective type family
    • In the expression: x
      In an equation for ‘test3’: test3 x = x
    |
191 | test3 x = x
    |           ^
```

The error that is given blames the fact that `Y` is not injective. Although this is true and `Y` indeed does not have an injectivity annotation, this does not matter. `Y` has one instance and thus it is injective. Furthermore, the error has a different cause: both `Y Bool` and `Y Float` are not reducible. The type inference system tries to match them which fails. Instead of the current error, the error should mention the fact that the applications are not reducible to ground types because no instances exist.

Finally, we discuss a piece of program obtained from *stack overflow*[1].

```haskell
type family X a = fa | fa -> a

convert :: X a ~ X b => a -> b
```

The type family `X` is injective in the argument `a`. The function `convert` represents the definition for injectivity as discussed in section 4.5.1. At first glance, it would appear that this definition would type check, as the injectivity for `X` is defined. At a second glance, there is no instance defined for `X` that would possibly be able to reduce $X\ a \sim X\ b$. The error given by GHC is thus as follows:

```
main.hs:162:13: error:
    • Could not deduce: a ~ b
      from the context: X a ~ X b
        bound by the type signature for:
                   convert :: forall a b. (X a ~ X b) => a -> b
        at main.hs:161:1-30
      ‘a’ is a rigid type variable bound by
        the type signature for:
          convert :: forall a b. (X a ~ X b) => a -> b
        at main.hs:161:1-30
      ‘b’ is a rigid type variable bound by
        the type signature for:
          convert :: forall a b. (X a ~ X b) => a -> b
        at main.hs:161:1-30
    • In the expression: x
      In an equation for ‘convert’: convert x = x
    • Relevant bindings include
        x :: a (bound at main.hs:162:9)
        convert :: a -> b (bound at main.hs:162:1)
```

[1]https://stackoverflow.com/questions/55543598/why-is-this-injective-type-family-not-actually-injective

```
    |
162 | convert x = x
    |             ^
```

The error is a mouthful and is confusing in many ways. First of all, it talks about rigid type variables. A definition that is not known by many programmers. It talks about how these type variables are bound, which could help but is not understandable for a novice programmer. The most interesting part of this error is the point where it mentions that $a \sim b$ cannot be deduced from $X\ a \sim X\ b$. Injectivity is accepted by the compiler but the most trivial injective function is not! The trick here is to add the instance `type instance X a = a`. Injectivity cannot be used to define instances. Furthermore, when this instance is added, the injectivity annotation becomes unnecessary as the constraint $X\ a \sim X\ b$ can be reduced by simply using left-to-right rewrite rules.

5.3.3 Nested type family applications

First, let us consider a piece of program in which type families are used in a nested way:

```haskell
type family H a b where
  H Int Int = Float
  H Float Float = Int

type family J a
type instance J Int = Int

wrongFunction :: H (J Float) (J Int)
wrongFunction = 1
```

When performing type inference, the system should first reduce `J Float` and `J Int` to simpler types before `H` can be reduced. In this situation, `J Float` does not have an instance that can reduce it, which is exactly what the type inference system should mention. GHC instead throws the following error:

```
main.hs:218:17: error:
    • No instance for (Num (H (J Float) Int))
        arising from the literal ‘1’
    • In the expression: 1
      In an equation for ‘wrongFunction’: wrongFunction = 1
    |
218 | wrongFunction = 1
    |                 ^
```

Instead of noticing that `J Float` cannot be reduced, GHC instead sees `H (J Float) Int` as a type, which again is the result of the fact that type families are seen as total functions. GHC misses an instance of `Num` for this new type because it should be the type of the value 1. This is *far* from the actual cause of the error. As a result, the programmer may not notice that no instance for `J Float` exists.

Next, consider the following contrived type family definitions and functions:

```haskell
type family H a b where
  H Int Int = Int
  H Float Float = Int

type family J a
type instance J Int = Int

f :: Int -> H (J Int) (J Int)
f x = x + 1

wrongF :: Bool
wrongF = f 2 == True
```

The function `f` type checks. `J Int` reduces to `Int` and `H Int Int` then reduces to `Int` which is the result type. Next, `wrongF` uses `f` in an equality, which naturally does not type check. The error is as follows:

```
main.hs:221:10: error:
    • Couldn't match type ‘Int’ with ‘Bool’
      Expected: Bool
        Actual: H (J Int) (J Int)
    • In the first argument of ‘(==)’, namely ‘f 2’
      In the expression: f 2 == True
      In an equation for ‘wrongF’: wrongF = f 2 == True
    |
221 | wrongF = f 2 == True
    |          ^^^
```

The first line of the error is definitely correct: `Int` could not be matched with `Bool` as required by `==`. The next lines could, however, confuse the programmer. The error tells us that it expected the type `Bool` and that the actual type is `H (J Int) (J Int)`. The first line shows the reduced type `Int`, whereas the third line shows its original version which is probably obtained from the type signature. The relationship between them might not be directly clear and may have gotten lost in the type inference process. Helium uses an algorithm called the *Binding Group Analysis* to determine the order in which certain pieces of code need to be type-checked. The idea is as follows: a function a may use a certain function b in its definition. For a to be type-checked, we need to have the type for b. Therefore, the type of b needs to be inferred before a. The process of obtaining this order is what Helium calls the *binding group analysis*. Type checking subprograms apart from each other has one disadvantage: the intermediate inference steps are lost when the correct type for a binding group has been found. Only the final type is used in the next binding group. The potential type family application steps may also get lost which results in the loss of valuable inference information which may help the programmer understand the error.

Chapter 6

Research questions

The main goal of this thesis is to improve type error diagnosis for type families in Haskell. Section 4 shows that the implementation of type families is not trivial and that type families become quite restricted to ensure termination and to keep it decidable. As a result, type families become less intuitive for the programmer. Section 5 shows that error quality concerning type families also suffers. Often errors do not point to the right cause, which results in confused programmers. Type family applications relate to their applied definitions but this relation is not always clear in type errors in GHC. All in all, the errors produced by GHC do not comply with the manifesto by Yang et al. [38] as discussed in section 3.3.1.

Helium in combination with Rhodium allows us to implement type inference of type families with the use of type graphs. Currently, these type graphs already support them. As a consequence, we are able to devise heuristics for them to improve type error diagnosis. We, therefore, want to answer the following main research question in this thesis.

Research question *How can causes of type errors involving Type Families be discovered and explained using heuristics?*

We aim to answer this question by improving type error diagnosis for all three versions of type families including associated type synonyms. The devised heuristics should be compatible with all heuristics that are currently defined for Helium. All versions of type families are currently not implemented in Helium. As a result, the static checks discussed in section 4 and the type inference for type families need to be implemented. Furthermore, we would like to keep track of type family application steps to be able to provide better error messages. The reason behind this is discussed in section 5.3.3. We thus pose two sub-questions that are needed to answer the main research question.

Sub question 1 *How can the different versions of type families be implemented to work concurrently in Helium?*

Sub question 2 *How can we keep track of type family application steps throughout the type inference process?*

Implementing the different versions of type families as one package allows the compiler and, more specifically, the type inference system to reason about why a certain type of type family has been chosen. Keeping track of the type family application steps also provides more information on the type error diagnosis process. By implementing type families in Helium and devising heuristics for them, we aim to improve the type errors shown and discussed in section 5. Some of these heuristics are based on the type family application steps that are stored.

Chapter 7

Implementation of Type Families in Helium

This section describes how Type Families have been implemented in Helium. The implementation follows the implementation of Type Families in GHC as close as possible. Because the implementation of the parser in Helium was straightforward, we omit its explanation from this report. We discuss how type family information is elicited from the Abstract Syntax Tree (AST) of Helium. Next, we discuss the data structures used and we discuss why the saved information is needed. Next, we provide a list of the static checks that have been implemented. For several of these, we need a unification algorithm. We discuss this algorithm and its possibilities. Furthermore, we discuss a new axiom that helps us to implement closed type families. Last but not least, we discuss how the X of the type inference engine is implemented for type families.

7.1 Information Elicitation

After parsing the type families, we store them in a convenient format so that we may easily reach all information needed for the static checks. We introduce two data structures. One in which we store Type Family *declarations* and one in which we store Type Family *instances*. The data structures are given in code snippet 7.1 and code snippet 7.2.

```haskell
data TFDeclInfo = TDI {
    tfdName :: Name -- name of the type family
  , argNames :: Names -- names of the arguments
  , tfdType :: TFType
  , injective :: Bool
  , injNames :: Maybe Names -- possible names that are injective
  , classIdx :: Maybe [(Int, Int)] -- possible indices of tf args linked to class args (ATS)
  , classDName :: Maybe Name -- possible name of the accompanying class
  } deriving (Show, Eq)
```

Code Snippet 7.1: Data structure for a Type Family declaration

The name of the type family, the names of their arguments and the type of the type family are stored respectively in `tfdName`, `argNames` and `tfdType`. The type of `tfdType` is defined as visible in code snippet 7.3 and determines what type of type family we are dealing with. The type family may be open, closed, an associated type synonym (ATS), or a type synonym. Furthermore, we store whether the type family is injective and in which arguments. A type family does not have to be injective. Therefore, this attribute is a `Maybe` type. Last but not least, we store the potential name of a type class and the indices of those variable names that are linked to certain variable names of a class. This information is only relevant when we are dealing with an Associated Type Synonym and is empty otherwise. This information is stored in `classIdx` and `classDName`.

```haskell
data TFInstanceInfo = TII {
    tfiName :: Name
  , argTypes :: Types
  , defType :: Type
  , tfiType :: TFType
  , priority :: Maybe Int -- priority inside closed type family
  , classTypes :: Maybe Types -- class types in case of ATS
  , classIName :: Maybe Name -- class name of accompanying class
  , preCompat :: [Int] -- compatible precomputations.
  , tfiRange :: Range
  , closedDeclName :: Maybe Name
  , varNameMap :: Maybe (Map Int Name)
  } deriving (Show, Eq, Ord)
```

Code Snippet 7.2: Data structure for a Type Family instance

In `TFInstanceInfo`, we store information obtained from `type instance`'s. Again, we store the type family name to which the instance belongs. Next, we store the *argument types (lhs)* in `argTypes` and the *definition type (rhs)* in `defType`. We also store the type of type family. The `priority` is only relevant for Closed Type Families and determines the order in which the instances are placed under the declaration. For associated type synonyms, we want to determine if types match for those variables that are equal in the class and ATS definition. We therefore save the types assigned to the ATS instance. `preCompat` saves the indices of other instances under the same closed type family declaration with which the instance is *compatible*. This information is used during the type inference process to determine if an instance may be used to apply a certain type family. Last but not least, we employ a `varNameMap` that contains the names used in the Type Family declaration related to their index. We use this to retrieve the names when we build the *injectivity annotation hint*.

```haskell
data TFType
  = Open -- open type family
  | Closed -- closed type family
  | ATS -- associated type synonym
  | TypeSyn -- type synonym
  deriving (Show, Eq, Ord)
```

Code Snippet 7.3: Data structure for the type of Type Family

The complete definitions of these data structures can be found in the file `TypeFamilyInfos.hs` in the repository of the Helium compiler.

7.2 Implemented static checks

We list the static checks for Type Families currently implemented in Helium in table 7.1. For each, we describe their goal and on which types of Type Families they apply. We do not consider type synonyms. We also omit a detailed explanation per static check as we already discuss them in section 4. The implementation of the static checks can be found in the file `TypeFamilies.hs` in the folder *static checks*.

7.3 Unification algorithm in Helium

For the compatibility check and the pairwise injectivity check, we need to perform different types of unification, as explained in sections 4.3.2 and 4.5.1. To this end, we implemented a unification algorithm in Helium that is able to perform *matching*, *unification* and *pre-unification* or a combination of matching and pre-unification, called *pre-matching*. The algorithm is heavily based on the standard unification algorithm by Robinson [27] and its implementation is similar in form to the (larger) one in GHC. The exact implementation of the algorithm can be found in the file `Unify.hs` situated in the type inference folder of the Helium compiler.

Name	**Restriction**	**Applies on**
No identical vars	No equal pairs of variable names may exist in a type family declaration	All
Injective variables	All variables in the injectivity annotation must be defined. Also goes for the result var.	Injective type families
Duplicate type families	A name for a top level declaration may only be used once.	All
ATS variable alignment	For ATSs, their arguments must align with their respective class instances.	ATSs
Undefined type families	Instances must always have a corresponding declaration	All
ATS not in class	ATS instances must always be defined in the corresponding class instance	ATSs
ATS not in correct class	ATS instances must be defined in the class instance corresponding to the one for which they were defined	ATSs
Type family defined under class instance	Non ATS type family instance situated under class instance.	Open / Closed
Open instance for closed	An open instance may not be created for a closed type family.	Closed
Wrong instance under closed	Only instances corresponding to the declaration may be present under a closed type family declaration	Closed
Instance Saturation	All arguments of a type family must be saturated	All
Var occurs check	All variables in the RHS must occur in the LHS	All
Termination guarantees	Non injective type family termination guarantee checks	Non injective
Compatibility check	Checks if there are pairs of overlapping instances	Open / Closed (warning)
Pairwise injectivity check	Checks if instances comply with the injectivity rules (using algorithm U)	Injective type families

Table 7.1: All implemented static checks for type family declarations and instances.

7.4 Axiom for Closed Type Families

To handle the ordered application of closed type families, we introduced a new type of axiom to the constraint environment of Helium: the *Axiom Closed Group*. The axiom is a wrapper around a set of type family axioms. It holds the name of the family that the axioms belong to and the axioms created from the instances belonging to that (closed) type family. This construct allows us to handle the axioms from a closed type family in an ordered fashion and makes it simple to perform the necessary *apartness* and *compatibility* checks between them. The set of axioms in the *Axiom Closed Group* is always ordered equally to the way the instances are ordered in the source code. Simply put, the *Axiom Closed Group* represents an ordered list containing the instances of a certain closed type family.

7.5 The X for Type Families in Helium

All the static checks discussed in section 4 and listed in table 7.1 make sure that the implementation of type families is type safe, decidable and make sure that the type inference process terminates. This section discusses how the X for Rhodium in Helium is implemented and why certain decisions were made. Some parts of the implementation remained unchanged with respect to the definitions given by Burgers et al. [4]. This section first discusses the extension of the constraint environment of X and then discusses the implementation of X per simplification rule as explained in section 3.2.3. For reference, the constraint environment is implemented in the file `RhodiumTypes.hs` and the X in `RhodiumInstances.hs`.

7.5.1 Constraint environment

The constraint environment, as described by Vytiniotis et al. [36] and implemented by Burgers et al. [4], is extended on two parts: the axioms and monotypes. Because type families are defined at top level, we extend the axiom definition $\mathcal{Q}$ with the following rule:

$$\begin{aligned} \mathcal{Q} &::= \ldots \\ &\quad \mid \forall[\bar{\alpha}].\ \tau \sim F\ \bar{\xi} \end{aligned}$$

where $\bar{\alpha}$ denote the touchable type variables and τ denotes the type that represents the type instance body. This type is subject to the restrictions posed by the static checks. $F\ \bar{\xi}$ represents the type family name and its arguments. ξ denotes a *type family free* type. This agrees with the restrictions imposed on type family instance arguments as described earlier in this section. This axiom is spawned when the programmer defines a `type instance` in the program.

Furthermore, the monotype definition is extended to incorporate type family applications in types:

$$\tau ::= \ldots \mid F\ \bar{\tau}$$

where F is the type family name and $\bar{\tau}$ represent the types the F is applied to.

7.5.2 Canonicalization

For type families, the canonicalization rule has the task to deal with type family applications that contain type family applications in their arguments. In other words, *nested type families*. Furthermore, the canonicalization rule implements one of the simplifications that injective type families allow. In

$$\begin{array}{lcll}
\mathit{canon}(tv \sim \tau_1\tau_2) & & & \text{Family in application} \\
\quad \text{where } \mathit{unfamily}(\tau_i) = (\bar{\alpha}_i, \xi_i, Q_i) & = & (\bar{\alpha}_1\bar{\alpha}_2, \{tv \sim \xi_1\xi_2\} \cup Q_1 \cup Q_2) & \\
\mathit{canon}(F\ \bar{\tau} \sim \tau') & & & \text{Nested type families} \\
\quad \text{where } \mathit{unfamily}(\tau_i) = (\bar{\alpha}_i, \xi_i, Q_i) & = & (\bar{\alpha}_1..\bar{\alpha}_n, \{F\ \bar{\xi} \sim \tau'\} \cup (\bigcup_i^n Q_i)) & \\
\mathit{canon}(F\ \bar{\tau} \sim F\ \bar{\sigma}) & & & \text{Injectivity} \\
\quad \text{where } \mathit{injArgs}(F\ \bar{\tau}) = (\bar{\tau}') & & & \\
\quad \text{where } \mathit{injArgs}(F\ \bar{\sigma}) = (\bar{\sigma}') & = & (\epsilon, \{\tau_i \sim \sigma_i \mid \tau_i \leftarrow \bar{\tau}', \sigma_i \leftarrow \bar{\sigma}'\}) &
\end{array}$$

Figure 7.1: Canonicalisation rules for type families (partly taken from [3])

figure 7.1, the canon rules for type families are given. The first two rules handle nested type families. Nested type families are handled by the *unfamily* function. Vytiniotis et al [36] call this procedure *flattening* and it is similar to the flattening procedure described in section 4.4.1. The rule is of the shape $\mathit{unfamily}(\tau) = (\alpha, \xi, Q)$, where τ may contain a type family application. Every type family application in τ is replaced by a fresh type variable β, henceforth known as a β-variable, while the shape of the constraints remains the same. The constraint Q contains equalities of the form $\beta \sim F\ \bar{\tau}$ with a fresh β for every type family $F\ \bar{\tau}$. Take as an example $\mathit{unfamily}(F\ \mathit{Int} \rightarrow G\ \mathit{Bool}) = (\{\alpha, \beta\}, \alpha \rightarrow \beta, \alpha \sim F\ \mathit{Int} \wedge \beta \sim G\ \mathit{Bool})$. The variables α and β are freshly introduced and replace the occurrences of the type families F and G.

The third rule handles injectivity. It first checks whether the type family at hand is injective. This is determined beforehand. The rule uses a function of the form $\mathit{injArgs}(F\ \bar{\tau}) = (\bar{\tau}')$ which obtains the injective arguments $\bar{\tau}'$ from a type family $F\ \bar{\tau}$. The rule then creates new equality constraints for each pair of injective arguments of the type family as per the definition of injectivity (section 4.5.1).

7.5.3 Interaction

Figure 7.2 shows the interaction rules as given by Burgers et al. [4]. Notice that all types on which the interact rule works must be *type family free*, as denoted by the use of ξ. The flattening of type families in equations opens up more possible interactions between those equations and other constraints [36].

$$
\begin{array}{lcl}
interact(\tau_1 \sim \tau_2, \tau_1 \sim \tau_2) & = & \tau_1 \sim \tau_2 \\
interact(tv \sim \xi_1, tv \sim \xi_2) & = & tv \sim \xi_1 \wedge \xi_1 \sim \xi_2 \\
interact(tv_1 \sim \xi_1, tv_2 \sim \xi_2) & & \\
\quad \text{where } tv_1 \in fv(\xi_2) & = & tv_1 \sim \xi_1 \wedge tv_2 \sim [tv_1 \mapsto \xi_1]\xi_2 \\
interact(tv \sim \xi, D\ \bar{\xi}) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) & = & tv \sim \xi \wedge D\ [tv \mapsto \xi]\bar{\xi} \\
interact(D\ \bar{\xi}, D\ \bar{\xi}) & = & D\ \bar{\xi} \\
interact(tv \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) \text{ or } tv \in fv(\xi_2) & = & tv \sim \xi_1 \wedge F\ [tv \mapsto \xi_1]\bar{\xi} \sim [tv \mapsto \xi_1]\xi_2 \\
interact(F\ \bar{\xi} \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & = & (F\ \bar{\xi} \sim \xi_1) \wedge (\xi_1 \sim \xi_2)
\end{array}
$$

Figure 7.2: Interaction rules (taken from [3])

The interaction rules introduce two new rules that involve (flattened) type families. The first interacts $tv \sim \xi_1$ and $F\ \bar{\xi} \sim \xi_2$. The rule allows the type variable tv to be substituted inside the type family arguments and definitions ξ and ξ_2. The second new rule and the last rule in figure 7.2 allows the interaction of two type family constraints that are applied equally. This should result in the fact that their definitions ξ_1 and ξ_2 should be equal, which is why the new constraint $(\xi_1 \sim \xi_2)$ is created.

We introduce one additional rule for injective type families. If the last rule in figure 7.2 fails, we may check if ξ_1 and ξ_2 are equal. If this is the case, we may introduce evidence that the injective arguments to F are equal. We may thus introduce those new equalities. We formalize this new rule as follows:

$$
\begin{array}{l}
interact(F\ \bar{\xi}_1 \sim \xi_3, F\ \bar{\xi}_2 \sim \xi_4) = (F\ \bar{\xi}_1 \sim \xi_3) \wedge \{\xi_1 \sim \xi_2 \mid \xi_1 \leftarrow \bar{\xi}_1, \xi_2 \leftarrow \bar{\xi}_2\} \\
\quad \text{where } equal(\xi_3, \xi_4)
\end{array}
\tag{7.1}
$$

7.5.4 Simplification

The simplification rules are very similar to the interaction rules as explained in section 3.2.3. We therefore shortly focus on the two rules that this simplification step introduces. These are visible in figure 7.3.

$$
\begin{array}{lcl}
simplify(F\ \bar{\xi} \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & = & \xi_1 \sim \xi_2 \\
simplify(tv \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) \text{ or } tv \in fv(\xi_2) & = & F\ [tv \mapsto \xi_1]\bar{\xi} \sim [tv \mapsto \xi_1]\xi_2
\end{array}
$$

Figure 7.3: Simplify rules for type families (taken from [3])

The first rule simplifies two type family applications with the exact same arguments. The only difference with the interact version of this rule is that the given, left-most constraint is not returned as it is not needed further in this phase of the simplification process. The same goes for the second rule.

7.5.5 Top-level react

Next to the canonicalization rule, the top-level react is the most interesting rule when introducing type families. As mentioned, type families introduce a new type of axiom in the form $\forall[\bar{\alpha}].\ \tau_Q \sim F_Q\ \bar{\xi}_Q$ which represents a `type instance`. This rule is used to react with type family applications in the constraint system. The top-level react rule for type families performs the following steps, given a constraint of the form $F\ \bar{\xi} \sim \xi$. Note that all types must have been made *family free*.

1. We loop over all axioms that are defined in the constraint system and check if, for axiom $\forall[\bar{\alpha}].\ \tau_Q \sim F_Q\ \bar{\xi}_Q$ and type family application $F\ \bar{\sigma}$, $F = F_Q$. If not, we consider the next axiom.

2. If $F = F_Q$, we will try to unify $\bar{\xi}_Q \sim \bar{\xi}$, where the variables $\bar{\alpha}$ are considered touchable and may thus be unified to another type. This allows us to apply a type application on a more general type instance.

3. If step 2 is successful, the resulting substitution θ is returned. The result of the top-level react call is then $\theta\tau_Q \sim \xi$ where we apply the substitution θ to the definition of the type family axiom ξ_Q. We thus replace $F\ \bar{\xi}$ with its definition.

4. If step 2 is unsuccessful, we return to step 1 and loop further over the known axioms. When no matching axiom is found after all have been considered, the constraint is left residual.

The above steps work well for open type families, as there is no order implied. For closed type families, we utilize the *Axiom Closed Group* as explained in section 7.4. Because type family names are unique, we can extend the axiom loop function to also search for axiom groups. When such a group is found, we only need to loop over the axioms inside it. Furthermore, because we know it represents an ordered set of axioms from a closed type family, we can check the axioms in order. This also allows us to perform the *compatibility* and *apartness* checks over the complete group as explained in section 4.4.1. Compatibility is computed before the type inference process.

Top-level improvement When the type family that we want to simplify is injective, and the above process fails, we may attempt to perform a *top-level improvement*. When, again, given an axiom of the form $\forall[\bar{\alpha}].\ \tau_Q \sim F_Q\ \bar{\xi}_Q$ and a contraint of the form $F\ \bar{\xi} \sim \xi$, we perform the following steps:

1. We attempt to *pre-match* ξ and τ_Q. Pre matching describes the combined process of matching and pre unification. In short, we use the pre unification algorithm (figure 4.1) while only keeping the types in one order. If pre matching fails, we return. Else, we obtain the resulting substitution *subst* and advance to the next step.

2. We apply *subst* on the axiom family $F_Q\ \bar{\xi}_Q$ and check if it matches with its original version. This is important because it ensures that injectivity is indeed correct [34]. Again, if it fails, we return.

3. At this point, we may create new evidence based on the injective properties of the type family. We apply *subst* over the injective arguments in $\bar{\xi}_Q$ to create $\bar{\xi}'_Q$. We then build a set of new constraints as follows:
$$\text{injConstr} = \{\xi'_Q \sim \xi \mid \xi'_Q \leftarrow \bar{\xi}'_Q, \xi \leftarrow \bar{\xi}\} \tag{7.2}$$

4. We return the set of constraints *injConstr* and $F\ \bar{\xi} \sim \xi$. This constraint may be improved using the constraints in *injConstr* but may also be left residual in some form after type inference is complete. This, however, introduces a subtlety. It may the case that $F\ \bar{\xi} \sim \xi$ may not be able to be improved at all. In this case we do not want to perform above steps on it again as this will not provide new information. We, therefore, annotate the constraint with the notion that top-level improvement was already applied. This prevents the type inference system from looping.

Chapter 8

Tracing Type Family applications: the Reduction Trace

Type family applications may introduce new type families. Furthermore, type families may occur as arguments to other type families in annotated types. It may therefore take several type family applications before a base type is reached. During these applications, several intermediate types are created that are not present in the source code. This is intentional but poses a problem if such an intermediate type may not be reduced further. The resulting type error will contain a type that is not present in the source code, which may confuse the programmer. Furthermore, when a type family is fully reduced and the resulting base type creates an error, it may also be hard to find the original type as the previous steps are lost after type inference completes (Section 5.3.3). To this end, we implemented a system that traces changes concerning type families. We carefully choose the word *change* as the trace should also mention changes to type family arguments as a result of injective properties. This section discusses what we expect from the so-called *Reduction Trace* in terms of its requirements. Furthermore, we discuss its implementation in the Helium compiler and the difficulties that arose from it. Last but not least, we conclude this section by discussing the effects of these difficulties.

8.1 Design requirements

We first list the requirements for the design of the Reduction Trace. In essence, we should adapt the current structure of the type inference system as little as possible. The trace should act as an extension of the existing system. We introduce the following requirements.

- **The Reduction Trace should be complete.** The trace should always find the reduction path from start to finish. Furthermore, no steps should be excluded. The first type in the trace should be the type that is present in the erroneous constraint. The last type should be present in the type signature that belongs to the function or other expression in which type inference goes wrong. That way, we may present a novice programmer with a clear step-by-step application sequence of the type family.

- **The trace should be presented in a type error clearly.** The steps that are part of the trace should be presented one by one with potential extra information that explains why the step was taken. For example, what instance was applied? How often was it applied and why?

- **The trace should be able to distinguish between different reductions.** The trace should contain standard *left-to-right applications* where an instance is simply applied. Next to that, it should also contain *argument injections* where argument information, possibly obtained by the injective properties of the type family, is inserted.

- **The trace should be able to be squashed.** This requirement is meant to improve the readability of the trace information in the type error. In some cases, a type family instance may be applied multiple times in a row. In this case, it is not helpful if each of the applications shows the exact information. In such a case, we can group those applications. In other words: we squash the trace. Squashing the trace is only viable when multiple *left-to-right applications* of the same instance are performed. Argument injections do not need this as the operation is always performed once with a certain set of constraints and is not linked to certain instances.

- **The Reduction Trace should be an extension of the existing system.** The trace should be gathered during the steps of the type inference system. We impose this requirement as we do not want to adapt the existing type inference system, which is based on a logic that is both sound and complete. This is a matter of not wanting to re-invent the wheel: we attempt to find a way to retrieve new information from the type inference process without changing its core.

The trace becomes a core part of type error messages in Helium, with a main role in the type error messages related to type families. We, therefore, use this section to discuss the reduction trace in a standalone fashion. The utilization of the trace will be discussed in sections 9 and 10, where the heuristics related to type families are introduced.

8.2 Implementation

This section explains and discusses the implementation of the Reduction Trace. We first explain the data structures we use and where they are situated in the type inference process. Next, we explain how we obtain reduction steps during type inference. Last but not least, we discuss how we build a complete trace from the steps obtained. The data structures can be found in `RhodiumTypes.hs` in Helium. Most functions built to work on the data structures can be found in `ReductionTraceUtils.hs`.

8.2.1 Monotypes and a reduction step

The constraint environment in Helium contains *monotypes* with which basic types can be expressed. The monotypes are defined as in figure 8.1.

Monotypes	τ	$::=$	α	variables
		$\mid$	$\tau_1\tau_2$	type variable application
		$\mid$	A	constants
		$\mid$	$F\ \bar{\tau}$	type families

Figure 8.1: The definition of monotypes in Helium

During type inference, type family monotypes can be applied to axioms defined for type family F. To build a trace of these applications, we extend the monotypes with a *reduction step*. The goal of this step is to save a potential application and thus reduction, of a type family. We add this step to monotypes, because every monotype may potentially be the result of a type family application. Furthermore, it allows us to quickly see if a type is the result from a type family reduction. This has its use when checking if a type family was to blame for a type error.

In the basis, a *reduction step* saves two types: the type before, and the type after the reduction. It also saves the type of reduction that was applied. We currently have two types of reductions: a *left-to-right application* and an *argument injection*. Both correspond to a way that a type family may progress towards a base type. The left-to-right application corresponds to when a type family is applied to an instance based on its arguments (hence left to right). The before type is the type family before application. The after type is the result after application. The argument injection step deals with changes to a type family when we may fill in argument variables due to injective properties. The before type is the type family without its new arguments. The after type is the same type family with its new arguments inserted.

The type of reduction may be extended by adding extra information about the reduction step. For example, in the case of a left-to-right application, we add the instance that was applied. The extra information may be shown when printing the reduction trace or used for potential other means, like finding reductions that apply the same instance to a type family.

Last but not least, we introduce the *Reduction Trace*, which is a list of reduction steps that describes how a type family is reduced throughout the type inference process. We formalize the data structures as implemented in Helium in figure 8.2.

The *location* in the LeftToRight reduction type holds the location of the instance that was applied in the source code. τ represents the *lhs* of the instance and τ' represents the *rhs*. For ArgInjection, τ and

ReductionType	rt	::=	LeftToRight $(\tau,\ \tau')$ *location*	left to right reduction
		$\mid$	ArgInjection $(\tau,\ \tau')$	argument injection
ReductionStep	rs	::=	Step $\tau\ \tau'\ rt$	one reduction step
ReductionTrace	$trace$	::=	$[(rs,\ Int)]$	the reduction trace

Figure 8.2: The reduction trace data structures

τ' represent the monotypes of the constraint related to the argument that was inserted. Last but not least, notice that the ReductionTrace saves an integer with every reduction step. This tells us how often a certain step was applied and is larger than one when the trace is squashed. We discuss squashing in section 8.2.3.

8.2.2 Obtaining steps during type inference

We obtain reduction steps at two points during type inference: during top-level reaction of a type family and at the interaction of a *variable constraint* with a *type family constraint*. The first saves a LeftToRight reduction step and the second an ArgInjection.

A LeftToRight reduction step is only saved when we find that an instance application succeeds. Given the constraint $F\ \bar{\xi} \sim \tau$ and *axiom* $\forall[\bar{\alpha}].\ \tau_Q \sim F_Q\ \bar{\xi}_Q$, we thus only do so if *map unify* $\{\xi \sim \xi_Q \mid \xi \leftarrow \bar{\xi}, \xi_Q \leftarrow \bar{\xi}_Q\}$ succeeds. If so, as explained in section 7.5.5, we create the new constraint $\tau'_Q \sim \tau$ where τ'_Q is the substituted version of τ_Q. We extend τ'_Q with the following step:

$$\text{Step } \tau'_Q\ (F\ \bar{\xi})\ (\text{LeftToRight } (F_Q\ \bar{\xi}_Q, \tau_Q)\ \text{loc}(\mathit{axiom})).$$

The step holds the initial and the result monotypes related to the type family in the obtained constraint. One may ask why it is needed to again save τ'_Q as it is the type that is extended. This is more a decision for convenience. We do not have to pass on the full monotype when constructing the trace in this way. The reduction type is always LeftToRight because we are at the only location where a left-to-right instance application can occur in the type inference engine. *loc* is a function that retrieves the location of the axiom. In Helium, the axioms are extended with the `TFInstanceInfo` obtained from the AST in which the location of the instance in the source code is saved.

An ArgInjection reduction step is built in a specific case during the interaction phase of the type inference system. It is introduced in the following rule:

$$\begin{array}{lcl} \mathit{interact}(tv \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & & \\ \quad \text{where } tv \in fv(\bar{\xi}) \text{ or } tv \in fv(\xi_2) & = & tv \sim \xi_1 \wedge F\ [tv \mapsto \xi_1]\bar{\xi} \sim [tv \mapsto \xi_1]\xi_2 \end{array}$$

This rule *inserts* new information obtained for arguments of a Type Family. As explained, this may be due to the injective properties of the type family. Whether this is the case or not does not matter as we always want to show any changes to the type family to the programmer when an error occurs. When we insert the reduction step, however, is quite subtle. First of all, we only do so if $tv \in fv(\bar{\xi})$ because we will only substitute an argument of F when this is the case. Furthermore, we do not add a reduction step when tv is a β-variable. As explained in section 7.5.2, such variables are only introduced during the *flattening* process which allows the type inference process to consider nested type family separately. When we encounter a β-variable in this rule, we are only undoing a flattening step which is introduced by the type inference system and should thus not be shown to the programmer. To conclude, we introduce the following reduction step to the type $F\ [tv \mapsto \xi_1]\bar{\xi}$ when $tv \in fv(\bar{\xi})$ and $tv \neq \beta$:

$$\text{Step } (F\ [tv \mapsto \xi_1]\bar{\xi})\ (F\ \bar{\xi})\ (\text{ArgInjection } (tv, \xi_1)).$$

The after type is the same type family as the before type with its arguments substituted by the first constraint given to the interact rule. The ArgInjection reduction type saves this constraint to be able to show it in a potential error later on.

8.2.3 Building a Reduction Trace

After the type inference process is complete, an arbitrary amount of types may have a reduction step inserted. The idea is that when such a type is part of an erroneous constraint, we may use the reduction trace built from the step inside the type to obtain more information about the error. If no extra info is obtained, we may as well show the trace to the programmer to provide more insight. To do this, we first must construct a trace from the step we may retrieve from the type. The trace is constructed in two phases, executed on a certain type τ.

Phase 1: following reduction steps We first follow the reduction steps one after the other. This phase checks whether type τ contains a reduction step. If it does, the step is simply added to the trace. We then obtain the *before* type from the step (type τ' in figure 8.2) and start anew. With every cycle, the step is placed after the steps obtained earlier. At some point, a type is reached without a reduction step. In this case, we inspect the type and check if it is a type application or a type family. These sorts of monotypes are recursive and thus contain monotypes that may in turn contain reduction steps. We consider these in the second phase. If the type does not contain recursive monotypes, we return.

Phase 2: diving deeper First of all, it is important to note that the reduction steps of recursive monotypes do *not* know that they were part of another type family or type application. This is due to the flattening that happens during type inference: every type family is considered separately. For the recursive monotypes, we first perform the steps like in Phase 1, where we obtain all reduction steps. These types may recurse, in which case we perform phase 2 as well. We, however, need to be careful when we obtain the trace. The steps inside the trace do not contain the type they were part of. We thus need to *insert* the steps back into the higher-level type. Let us show this situation with an example. Let us have the following type families

```
type family H a b where
    H Int Int = Float

type family J c where
    J Int = Int
```

and the type `H (J Int) (J Int)`. During type inference, this type is flattened to `H beta1 beta2` with constraints `beta1 ~ J Int` and `beta2 ~ J Int`. `J Int` reduces to `Int` but does not know that it was initially part of `H`. Luckily, inserting a trace back into a higher level type is not difficult: we loop over every step in the trace and encapsulate the *after* and *before* types in the higher level type. To be able to do this, we need to know what the argument was that the type belonged to. For type families, we therefore keep track of which arguments were already processed. We therefore also choose to present the traces within these arguments from left to right.

Last but not least, we discuss the possible length of a trace. Overall, a trace is limited in length. In many type family applications the amount of nesting is limited and there are often not many applications needed. However, there are cases where a trace might become too long. Furthermore, if we want to show a trace in a type error, we would want it to be concise as otherwise, we may risk that the error becomes unreadable. We, therefore, add the possibility for the trace to be *squashed*. In short, this means that we merge two steps in the reduction trace when we consider them *equal*. But when are two steps equal? The current implementation defines equality of two steps as the equality of their ReductionType. Note that this includes the equality of possible arguments to the ReductionType in question. For example, the LeftToRight reduction type also requires the applied instance and its location to be equal. Let us make the idea more concrete be introducing an example. Let us have the following type family:

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

and the type `Loop [[[[Int]]]]`. It is clear that the result type after type inference is `Int`. However, it takes *five* top-level react phases to reach this conclusion of which *four* use the exact same instance. Showing these four steps separately is unnecessary as it makes the error message fairly large and shows no new information. We, therefore, squash these four steps and instead mention in the error how often the instance was used. The *after* and *before* types of the new Step become the after step of the first step and the before step of the last step in the sequence.

8.2.4 Showing a Reduction Trace

Now that we have a Reduction Trace, we need to define how to show it. To this end, we reuse functionalities that Helium already possesses. A reduction trace is added to the type error similar to how a hint is added. A hint may be added to any error message, just like we want a reduction trace to be added to any error message. However, it is not needed for every error message that Helium is able to display. We, therefore, want to decouple the reduction trace from the error message to allow the maintainer of the compiler to decide where a reduction trace may be added. Type constraints in Helium are extended with the so-called *ConstraintInfo* structure that allows us to add any property to a constraint. For reduction traces, we add the *WithReduction* property, which allows us to add a reduction trace to a constraint at any moment. This reduction may (or may not) be shown in an error message later.

Currently, the reduction trace is shown as an enumerated list of steps. How a step itself is shown depends on the type of the step. The implementation in Helium allows that the show instances may be different per type of reduction. This makes it simple to possibly add more types of reduction and, more importantly, to perfect the way a certain step is shown to the user. To show the way a LeftToRight reduction step is show, we revisit the type family `Loop` from the previous section.

```
1.   Applied : Loop a = a
     From    : (149,5)
     Step    : Int <- Loop Int
     Reason  : left to right application
     Amount  : 1 time
2.   Applied : Loop [a] = Loop a
     From    : (148,5)
     Step    : Loop Int <- Loop [[[[Int]]]]
     Reason  : left to right application
     Amount  : 4 times
```

Notice that one step tries to tell the story of what happened. A certain instance was applied, which is situated somewhere in the source code, that resulted in a certain change to the type. Furthermore, it mentions the reason why this change was possible. Last but not least, notice that the second step was applied four times. The steps are ordered in such a way that each step comes closer to the original type family. We chose this ordering because the first step is now closed to the origin of the potential error in which the trace may be shown. From there, the programmer may read as far back as he or she likes. To show an example of an ArgInjection reduction step, we introduce the following type family:

```
type family Foo a b c = r | r -> c
type instance Foo Char Char Char = Bool
type instance Foo Float Bool Int = Int
```

Notice that the type family is injective in the third argument. We introduce the following constraint: `Foo t v t ∼ Bool` and obtain the following trace:

```
1. Applied   : t ~ Char
   On        : Foo t v t ~ Bool
   Step      : Foo Char v Char <- Foo t v t
   Reason    : argument injection
   Amount    : 1 time
```

Because we were able to find proof for `t ∼ Char` due to the injective properties of `Foo`, we were able to 'fill in' the `t`. Instead of an instance that was applied, we may now say that new proof in the form of constraint was. Notice that we do not have a location now, as that would be strange for a constraint. The fact that we show constraints in this trace step makes the error message that it might be included in *mechanical*. This criteria is part of the manifesto by Yang et al. [39] and states that no information from the type inference process should be shown. In this case, however, we deem it necessary because the constraints provide valuable information about what happens to the type family. This is the goal of the trace. We show more examples of the reduction trace when we discuss the heuristics built for type families.

Although the trace is squashed, we see from the examples that showing a trace takes up quite some room. We deem this necessary to make sure that it is readable and that the reduction step takes is clear. However, we recognize that an advanced programmer with lots of experience in type families may not need this extended trace. We, therefore, introduce the option to omit the trace from any type error that may occur. Currently, programmers may add the `--showtrace` argument to the Helium command to notify the compiler that they would like to see the full trace. This improves the quality of the error message based on the *succint* criteria from the manifesto by Yang et al. [39]. An expert programmer may not use the argument to get rid of the trace, while novice programmers may add the argument to get a better understanding of what happened with the type families throughout the program.

Last but not least, we recognize that showing a list of steps may not be the best idea with respect to the trace. Another way to show the trace is to build a graph-like figure that may link types (sometimes within types) with each other to represent how type families were reduced. It is, however, cumbersome to build such a figure in a pure text-based environment. We, therefore, chose to not explore the idea further.

8.3 The Curse of Non-linearity and other difficulties

At first sight, the implementation of the Reduction Trace seems valid and it works for the examples given. However, there are cases in which it fails to build a trace from start to finish. In some cases, it even fails to build a trace at all. In this section, we discuss several points of difficulty that haunt the Reduction Trace. Most of these difficulties are related to the process of obtaining the reduction steps during type inference. More specifically, type inference is not necessarily a linear process, where the current implementation of obtaining reduction steps assumes that it is.

8.3.1 Order of phases

The phases of building a reduction trace as described in section 8.2.3 are executed in a fixed order. Only when a type does not contain a reduction step, do we consider the possible traces in its recursive arguments. However, a type family instance may contain a type family on its right-hand side. After application, we may thus have a new type family inside a type which may be reduced to another type. As a result, we have a type that has a reduction step and also contains a recursive type with a reduction step. In this situation, the current implementation falls short. The order of phases should become more subtle.

8.3.2 Order of applications of a nested type family

When the type inference system type checks a nested type family, it first flattens it introducing β-variables in the arguments of the type family and separate constraints for the type families that were first in the earlier type family its arguments. Intuitively, one would expect that the type families that were positioned in an argument of their parent are considered first. Linearly, from nested families to their parents. In this sense, the constraints generated during flattening are first considered and (possibly) reduced during the top-level react phase. Afterward, the result types of the nested type families may be reinserted in the top-level type family which may then reduce. In many cases, the application of type families follows this linear idea. However, there are situations in which this is not the case. We explain these using an example:

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

type family Id c where
    Id c = c

tfconst :: a -> b -> Const a (Id b)
tfconst x y = x
```

We have two type families which represent the value-level funtions `id` and `const` of the type level. The function `tfconst` contains a type error because `Const a (Id b)` reduces to `b` and `x` has type `a`. We are however more interested in how the type family is reduced. Let us go over it step by step.

1. `Const a (Id b)` is flattened. We obtain the type `Const a beta1` and the new constraint `beta1 ∼ Id b`.
2. `Id b` is reduced to `b` and we obtain the constraint `beta1 ∼ b`.
3. `Const a beta1` is reduced `beta1`.

4. `beta1` is substituted with `b` in the constraint `beta1 ~ a` to create the constraint `a ~ b` where we return an error.

Steps two and three violate the order in which most reductions take place. The top-level type family is reduced before the reduced nested type family is reinserted. As a result, `b` only contains the step that it was reduced from `Id b`. This situation only happens in case we have very general type families like `Const` and `Id` that may be applied with any type and thus also with type variables. β-variables are type variables and may thus be used to apply the instance. In situations like this, the trace becomes incomplete as the reduction from `Const` has been forgotten during the type inference process.

We designed a fix by restricting top-level reaction that a type family may only attempt a reaction if all its arguments are *beta free*. In other words, all possible nested type families inside a type family must be considered first before the top-level family is reduced. In case a nested type family cannot be reduced, the top-level family will also stay unreduced. We argue that this does not break the *completeness* and *soundness* of the type inference system. The system remains complete as we do not allow fewer type families to be type-checked. If a type family is fully reducible, the type inference system will still do so, albeit in a different order. This also argues that soundness remains. In case a type family is not (fully) reducible, the type inference system will still return an error. We found, however, that in some cases, the error is a bit more general. Consider the following example:

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

type family Id c where
    Id Int = Int

tfconst :: a -> b -> Const (Id a) b
tfconst x y = x
```

`Id` is now less generic and only allows `Int` as an argument. As a result `Id a` in the type signature will not be reducible. In case we still allowed β-variables to be in the type family arguments during the top-level react phase, `Const` would still be reducible. We would get an error that tells us that `Id a` is not reducible. In our new situation, `Const` will not be reduced, as it will remain in the form `Const beta1 b` after flattening. After substitution, the type inference system will tell us that `Const (Id a) b` in total is not reducible. We argue that this is not a problem as the situation is very specific. Furthermore, as we will see when we discuss the designed heuristics: the type to blame is still inside the type and can be blamed.

8.3.3 Erroneous substitutions and more order issues

From de thesis report by Burgers [3], we quote the following:

> "We do not give any guarantees about the correctness of the substitution if the type graph is inconsistent."

In other words, we may not assume that the substitution is correct after the type inference system is done and an error occurs. During the creation of the trace, we try to substitute some type variables to check if the result type has a trace that we may potentially explore. This turned out to be dangerous because it can cause looping. Especially when the substitution resulting from type inference is certainly not correct. If the type variable we substitute is also present in the before type, we end up in a loop because we eventually consider the type variable again. In the current implementation, we built in a safety measure that stops the creation of the trace after a certain amount of going into recursion. This amount is currently set to 50. We are therefore not able to guarantee that a reduction trace is created when an error occurs related to a (partly) reduced type family. This situation may occur when there are two type families present in the constraint system that are exactly the same. Like in the following example.

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

```
type family Id c where
    Id Int = Int

tfconst :: (Id a) -> b -> Const (Id a) b
tfconst x y = x
```

Even though `Id a` is not reducible, the following rule is applied during type inference:

$$interact(F\ \bar{\xi} \sim \xi_1, F\ \bar{\xi} \sim \xi_2) = (F\ \bar{\xi} \sim \xi_1) \wedge (\xi_1 \sim \xi_2)$$

At a certain point during the type inference process, we have two constraints: `Id a ∼ x1` and `Id a ∼ beta2` where `x1` is the type variable related to the argument `x` in the code and `beta2` is a variable introduced after flattening. When we apply the rule above, we obtain the constraint `x1 ∼ beta2`. Eventually, this constraint interacts with `Const beta2 b ∼ x3` to form `Const x1 b ∼ x3` which can reduce because it is *not* a β-variable. `x3` is the type variable related to the `x` in the body. At some point we end up with the constraint `x1 ∼ b` which creates the substitution $[x1 \mapsto b]$ and lets the creation of the reduction trace loop. This substitution is of course incorrect and is only created because the constraint system itself is incorrect. The fact that `beta2` is substituted for `x1` however also poses problems because it negates the fix we discussed in section 8.3.2. The order of applications of type families again becomes non-linear and poses the same problems as before.

8.4 Conclusion

We introduced the *Reduction Trace* to be able to follow the type family applications during type inference. The trace may be shown in a type error to provide more information to the programmer when a type error occurs that has to do with a type family. We were able to gather steps of the Reduction Trace during type inference without touching the core functionalities. We, however, found that the approach we chose fails in more complicated constraint systems where the non-linear nature of type inference becomes more clear. When referring back to the design requirements posed in section 8.1, we were able to satisfy all but one. The current implementation of the Reduction Trace is not always complete because it is not always possible to construct a full trace in a more complicated constraint system.

Chapter 9

The Type Family Reduction Heuristic

As explained in section 3.3.6, the task of a heuristic is to indicate the likelihood that a certain error has a specific cause. The goal of the heuristic that we discuss in this section is to indicate the likelihood that the error was caused by an erroneous reduction of a type family. This heuristic is a *voting heuristic* which means that it is a *selector* that competes with others by assigning a weight that represents how sure it is that a certain constraint is to blame. This section introduces a new selector or voting heuristic that selects a constraint when it is sure it is to blame for an erroneous type family reduction sequence. We first describe what the design should look like. Next, we discuss the implementation of the heuristic and certain choices that were made in the process. Furthermore, we discuss the quality of the error report that is shown after the heuristic wins the selection process. Last but not least, we discuss the interaction that the new heuristic has with other, already existing, heuristics. In Helium, the exact implementation can be found in `TypeFamilyHeuristics.hs` situated in the `HeuristicsOU` directory which in turn is part of the `StaticAnalysis` directory.

9.1 Design Requirements

We list a few important requirements that the design of the type family reduction heuristic should abide by. In essence, the new heuristic should fit easily within the existing heuristic environment and should not clash with earlier, correctly working, heuristics. This interaction is discussed later. Furthermore, we introduce the following design requirements.

- **The heuristic should utilize the Reduction Trace.** First of all, we would like to show the reduction trace in a potential error report when this heuristic wins the selection process. More importantly, we would like to use the reduction trace to expand the number of erroneous constraints found to be related to a type family reduction. We can do this by following the reduction trace of a type back to the beginning.
- **The heuristic should handle nested type families correctly.** In case an erroneous type family contains type families in its arguments (nested type families), these families are (partly) to blame for the fact that the type family was not reducible. The heuristic should consider these families during the blaming process.
- **The heuristic should attempt to find an explanation of why the error occurred.** This should be in the form of a hint given in the error report. In this thesis, we designed two specific hints to two specific types of errors that a programmer can make. Furthermore, if a type family in the erroneous constraint was not reducible without a specific known reason, it should be pointed out.

In short, the heuristic should, when it fires, attempt to gather as much information as needed to construct the resulting type error message. The heuristic should also invoke the correct *blueprint* for the error message. This blueprint determines the form of the error, only to be filled by the information gathered by the heuristic. The implementation for this heuristic needs a potential reduction trace and a set of hints to fill in the blueprint.

9.2 Implementation

We discuss the implementation of the heuristic by considering the steps it takes to reach a verdict on whether type family reduction may be the cause of the error. At some point, the heuristic defines different steps based on the exact form of the erroneous constraint that it receives. We therefore first discuss two hints that were defined before we bring them into context in the heuristic. The two hints we defined are the *permutation hint* and the *apartness hint*.

9.2.1 The permutation hint

The permutation hint aims to consider the arguments of a given type family. It is able to fire upon a constraint of the following specific form: $F\ \bar{\sigma} \sim \tau$. When given to the heuristic, we know that $F\ \bar{\sigma}$ was not reducible. Furthermore, we know that the constraint system expects that it was reduced to τ. It may be the case that the programmer messed up the order of the arguments given to the type family. The permutation hint, therefore, tries if a *permutation* of the current order of arguments in $\bar{\sigma}$ may fix the constraint. We make the idea more concrete with the following example

```
type family Swap a b
type instance Swap Int Float = Int

f :: Swap Float Int
f = 4
```

where it is clear that the arguments are ordered incorrectly. The hint then provides us with the following message:

Changing "Swap Float Int" to "Swap Int Float" helps to remove this type error.

It tells us that this action helps to remove the current type error. In this case, it even completely fixes the error. However, in larger, possibly nested, type families there may be more problems that need to be solved before a certain type error is removed. We thus can't tell the programmer that the action *fixes* the error. Notice that the hint always blames the type family in the *type signature*. Apart from the type signature, we may in some cases also blame the body of the function or the instance of the type family. We may, for example, say that `Swap Int Float` in the *instance* should have its arguments swapped. This is cumbersome, especially when the instances of the type families are scattered over multiple code files. Furthermore, changing around the arguments in the instance may break other functions that are correctly implemented using the type family. We, therefore, choose to keep the hint local and blame the type signature. It is often the case that the programmer writes the body of the function first and the type afterward. We may therefore assume that the body is correct and the type signature needs fixing.

Implementation The implementation of the permutation hint is straightforward. First, the permutations of the arguments of F are constructed. We recognize that constructing a set of permutations is a costly operation. It is however rare that a type family is defined with more than three arguments in which case the operation does not take too long. Next, the hint loops over these and constructs the new constraint $F\ \bar{\sigma}' \sim \tau$ where $\bar{\sigma}'$ is the permutation. This new constraint is given to the type inference system to check whether it is now correct. If it is, we construct the hint to form a message like the one above and return If the constraint is still incorrect, we try the next permutation. If all permutations fail with respect to these steps, no permutation hint is given. In case multiple permutations succeed, the current implementation provides the first one found as hint. The others are discarded. Another option is to provide all possible permutations in the error. This allows the programmer to choose the permutation he or she might need. We argue that it is hard to determine which one of the permutations the programmer meant when writing the code. We are not even sure if a permutation of the arguments is what the programmer meant to do. However, in the future, it might be better to provide all possible permutations at once instead of only the first one found.

9.2.2 The apartness hint

The apartness hint is only relevant in case a *closed* type family was not reducible. In some cases, the culprit may be the *apartness check* (Section 4.4.1) that is performed when applying closed type family instances. This may lead to confusing situations, like in the following example.

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

loop :: Loop [c]
loop = True
```

The body of the function `loop` is not relevant. At first sight, it seems like the type family in the type signature is fully reducible. However, it gets stuck after it obtains `Loop c` from applying the first instance. To be able to apply the second instance, the type inference system performs the apartness check (Section 4.4.1) and finds that `Loop c` is not apart from `Loop [a]`. `c` may be instantiated as `[c]` and therefore the instance `Loop a = a` may not be applied. The hint provides the following message.

> type ”Loop c” is not apart from instance ”Loop [a]” at (148,5).

`Loop c` is an intermediate type that is the result of a type family application. Where this type came from, becomes clear when this hint is put into the context of the type error message. When focussing on the hint alone, notice that it specifically mentions which instance was not apart and mentions its location so that the programmer may easily find it. The compiler does not decide whether the instance or the type signature is to blame for the error. It is hard to find which of the two is to blame because the error may arise because of many reasons. We, therefore, leave this for the programmer to figure out and show the hint as a *probable cause* rather than a fix.

Implementation The hint first checks if the given type family is a closed one. If not, we may immediately return as there is no way the apartness check can be an issue. If the type family in question is closed, we try the specific top-level reaction again, this time with a flag that we want more information returned. It also skips any top-level improvement as we do not need it. If the apartness check fails, the top-level react phase now returns the instance and the location of the culprit. This information is then used to construct the hint as given above. If the top-level react phase returns nothing, we do not construct the apartness hint.

9.2.3 The heuristic

Now that we have discussed the hints that may be given, we continue to how the heuristic itself is set up. First of all, the heuristic obtains the constraint from the path that is currently considered (one at a time), henceforth known as the *path constraint*. Furthermore, the heuristic obtains the complete path so that it can obtain the erroneous constraint, known henceforth as the *erroneous constraint*. Type families may only occur in explicitly declared types and are in no way inferred. The path constraint may therefore only be a constraint obtained from a *type signature* or other *type annotation* from the source code. In Helium, such a constraint is encoded as an *Instantiation Constraint*. The heuristic thus checks whether the path constraint is encoded as such. If not, we return. The main idea of the heuristic is that the erroneous constraint is then checked against the path constraint. However, the erroneous constraint has different forms that are each compared differently to the path constraint. We introduce and discuss each form separately. Before we do so, we introduce the constructor in which the information gathered by the heuristic, is saved. The constructor is part of the *Property* data structure in *ConstraintInfo* and looks as follows.

$$\begin{aligned}\text{Property } p ::= & \ldots \\ & \mid \text{TypeFamilyReduction } (\mathit{Maybe}\ \tau_1)\ \tau_2\ (\mathit{Maybe}\ \tau_3)\ \tau_4\ \mathit{Bool}\end{aligned}$$

The monotypes τ_1 and τ_2 and similarly τ_3 and τ_4 form pairs. Each pair represents the *original* type (uneven) and the obtained type (even) from the left and right sides of the constraint respectively. τ_1 and τ_3 are encapsulated in a Maybe. We do this because a type does not necessarily need to be reduced from a type family. The boolean is false if the constraint contains a type family that was not fully reducible and true otherwise. Notice that a potential reduction trace is not saved in this constructor. We omit the trace because the heuristic saves them in the *WithReduction* constructor (also part of Property) to allow the trace to be used more generally. We may now introduce and discuss each of the forms of the erroneous constraints.

1. $F\ \bar{\sigma}_1 \sim G\ \bar{\sigma}_2$. When we have two type families that need to be equal, we know that both were not fully reducible. We also know that we cannot perform the permutation hint as we do not know what the constraint system expects one type family to reduce to. We thus firstly substitute any β-variable still present in both σ_1 and σ_2 to ensure that we have the complete original type. We then obtain the *reduction traces* for both F and G and obtain the *last type* in the traces. If no trace was found, the last type becomes the original type. Furthermore, the heuristic builds the apartness hint, which can be done for each type family separately and does not need any extra information, unlike the permutation hint. However, because the type families may contain nested type families, we must try to build this hint for these as well. To this end, we recurse in the arguments of $\bar{\sigma}_1$ and $\bar{\sigma}_2$ and build a *nested apartness hint*. The hints resulting from a more deeply nested family take precedence because these are a cause of the fact that the parent family was not reducible. Note that multiple hints may be formed. These are grouped and shown as one. Last but not least, we check if one of the *last types* is part of the type annotation represented in the path constraint. If it is, the heuristic knows it has found the constraint that it can blame. It adds the TypeFamilyReduction constructor, the possible hints, and the possible reduction traces to the constraint info and returns.

2. $F\ \bar{\sigma} \sim \tau$. In case we have a constraint formed this way, we have more information on what the type family should have reduced to in terms of a base type. This base type is τ. We also know that we are still dealing with a non-reducible type family. In this case, we can perform both the permutation and apartness hints. Furthermore, because $\bar{\sigma}$ may contain nested type families, we must again apply the hints recursively. For the apartness hint, this happens in the same way as described in the previous case. For the permutation hint, the situation becomes difficult. To explain this, we introduce the following example program.

   ```
   type family Perm a b
   type instance Perm Float Int = Int
   type instance Perm Int Int = Char

   p :: Perm (Perm Int Float) Int
   p = 'N'
   ```

   Notice that, if `Perm Int Float` becomes `Perm Float Int`, the type family may reduce and the error is solved. We know this because `Perm (Perm Int Float) Int ∼ Char` and thus, based on the instance, `Perm Int Float ∼ Int`. Notice that we use injectivity here to find more information about the nested type family. We must, however, be careful because the type family is not annotated as being injective. Using τ, we filter the type family instances on the *rhs*. If only one set of arguments is returned, and thus only one instance complied, we may continue to recurse for the permutation hint. Otherwise, we abort recursion because we can never be certain which instance to choose. For the rest, the heuristic continues similar to constraint form 1: it obtains the trace from $F\ \bar{\sigma}$ and τ and determines, based on the last type from $F\ \bar{\sigma}$ if it obtained the correct path constraint. If so, we build the same constructor and return.

3. $\tau_1 \sim \tau_2$. A constraint of this form is interesting because there is no type family present. However, the erroneous constraint may very well be the result of a fully finished reduction. In this situation, the reduction trace plays its part. In case τ_1 or τ_2 or both have a trace, we may find that the source code was erroneous. We, therefore, try to obtain the last type from the traces, if they exist, and perform the permutation hint on them. Notice that it does not make sense to perform the apartness hint here, as there are no unreduced type families. We showcase the extra information that the reduction trace brings with the following example.

   ```
   type family Perm a b
   type instance Perm Float Int = Int
   type instance Perm Int Float = Char

   p :: Perm Float Int
   p = 'N'
   ```

   The type `Perm Float Int` is reducible to `Int` which results in the erroneous constraint $Char \sim Int$. The reduction trace sees that Int is reduced from $Perm\ Float\ Int$ and sees that swapping the arguments fixes the error. The hint thus becomes:

Changing "Perm Float Int" to "Perm Int Float" helps to remove this type error

Again, the rest of the heuristic is straightforward where we check if the types are part of the path constraint.

9.2.4 Choosing a graph modifier

In the heuristic system of Helium, a *graph modifier* is a function that adapts the graph in such a way that the considered type error is removed from the constraint system that the graph represents. The specific type of graph modifier that is applied is returned by a heuristic. A more detailed explanation can be found in the thesis by Burgers [3]. A graph modifier is applied after a heuristic is chosen to blame a certain error. The resulting graph may contain other errors that were in the original constraint system that may in turn be considered and are considered by the blaming process. This improves the comprehensiveness of the error reporting. For this heuristic, we decided to keep the standard graph modifier, which removes the path constraint from the graph. In our case, the path constraint contains the type signature or type annotation in which the problematic type family resides. Removing it has a downside: any other errors present in the type signature vanish after the resulting graph is simplified by the type inference system. The error reporting thus becomes not as comprehensive as we would have liked.

We experimented with trying to substitute the type family in the type signature with the type that was inferred. This worked to some extent but could introduce other errors that were a result of the initial error related to the type family. As this error is not directly introduced by the programmer and does not represent the main cause, this is not the behavior that we wanted. As a result, the standard modifier is used and we accept the loss of comprehensiveness.

9.2.5 The error construction

After the heuristic passed its judgment on the error path, Rhodium decides if its result is the most likely location for the error. If it is, the constraint info, adapted by the heuristic, is used to construct an error message. In this section, we discuss the structure of this message with the use of a few code examples. We first revisit the Perm example from earlier.

```haskell
type family Perm a b
type instance Perm Float Int = Int
type instance Perm Int Float = Char

returnN :: Perm Float Int
returnN = 'N'
```

Type checking the function `returnN` returns the following error message.

```
(270,1): Type error with type family reduction in explicitly typed binding
 could not match           :
   type                    : Char
   with                    : Int
     reduced from          : Perm Float Int
 in definition             : returnN
   declared type           : Perm Float Int
   inferred type           : Char
 possible fix              : Changing "Perm Float Int" to "Perm Int Float" helps
                             to remove this type error
 full reduction            : 1. Applied : Perm Float Int = Int
                                From    : (266,1)
                                Step    : Int <- Perm Float Int
                                Reason  : left to right application
                                Amount  : 1 time
```

First, notice that the title of the type error mentions in what type of language construct it resides. In this case an 'explicitly typed binding'. It also provides the location. Next, it mentions the constraint that went wrong: the type inference system cannot match type `Char` with `Int`. What is new in this error is that it potentially mentions from what type family the type was reduced from. In this case, `Int` was reduced from `Perm Float Int` which is present in the type signature. This information is obtained by the heuristic in the TypeFamilyReduction constructor and is shown here. The constructor created by the heuristic looks as follows

$$\text{TypeFamilyReduction } Nothing\ Char\ (Just\ (Perm\ Float\ Int))\ Int\ True$$

where *Char* was not reduced from a type family and *Int* came from *Perm Float Int*. The type family was fully reducible and we, therefore, have *True* as the last argument. Moving on, we see that the error further specifies the location of the error by specifying the definition in which it occurred and by specifying the differences between the declared type (in the type signature) and the type inferred by the type inference system. We add this to provide context to the erroneous constraint mentioned at the start of the error. Next, we notice the first hint in the context of the type error. The hint is a *possible fix* generated by the permutation hint. Its suggestion is valid as `Perm Int Float` would reduce to `Char`, solving the type error shown above. This hint is added to the constraint info through the *WithHint* construction, which was already implemented in Helium. Last but not least, the error shows the *full reduction* of *all* types in the erroneous constraint. In this case, we only have one as only one type was reduced from a type family. In other cases, there may be two reductions present. The full reduction part of the type error shows the reduction trace in the way that was explained in section 8.2.4. In this specific case, the reduction is quite simple and contains only one step.

The error message may provide multiple hints, like in the following contrived example program

```
type family Perm a b
type instance Perm Float Int = Int
type instance Perm Int Float = Char


returnN :: Perm (Perm Int Int) (Perm Float Float)
returnN = 'N'
```

from which the following error occurs.

```
(270,1): Type family in explicitly typed binding was not fully reducible
 could not match       :
   type                : Perm (Perm Int Int) (Perm Float Float)
   with                : Char
 in definition         : returnN
   declared type       : Perm (Perm Int Int) (Perm Float Float)
   inferred type       : Char
 probable causes       : "Perm Int Int" is not reducible. No matching instance
                         was found
                         "Perm Float Float" is not reducible. No matching instance
                         was found
```

The two nested type families in the type are not reducible. Both hints do not apply so, instead, the error points out the type families as culprits because no applicable instance was found for them. Furthermore, notice that the error does not blame the top-level type family. We do not have information on whether it would have been reduced because it has unreduced type families as arguments. We thus choose to not mention it. Lastly, note that no full reduction is present. This is because no applications have been performed. This shows that the heuristic also fires if a type family is not reduced at all.

Last but not least, we consider `Loop` type family again.

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

g1 :: Loop [c]
g1 = True

g2 :: Loop [Int]
g2 = True
```

We discussed the apartness hint that occurs when we type-check function `g1`. The hint shown in the context of the type error looks as follows.

```
(152,1): Type family in explicitly typed binding was not fully reducible
 could not match             :
   type                      : Loop c
     reduced from            : Loop [c]
   with                      : Bool
 in definition               : g1
   declared type             : Loop [c]
   inferred type             : Bool
 probable cause              : type "Loop c" is not apart from instance "Loop [a]"
                               at (148,5)
 full reduction              : 1. Applied : Loop [a] = Loop a
                                  From    : (148,5)
                                  Step    : Loop c <- Loop [c]
                                  Reason  : left to right application
                                  Amount  : 1 time
```

In case of function `g2`, apartness is not an issue. After `Loop [Int]` is reduced to `Loop Int`, `Int` is apart from `[a]` and thus the instance `Loop a = a` may be applied. The error we obtain is that Int does not match with Bool, inferred from the body. The complete error is shown below. Notice that we now have a larger reduction of two steps and that the error thus becomes quite large. The reason why a trace is potentially squashed should now be clear. Last but not least, notice that the different steps are neatly positioned under one another and are numbered in order.

```
(155,1): Type error with type family reduction in explicitly typed binding
 could not match             :
   type                      : Bool
   with                      : Int
     reduced from            : Loop [Int]
 in definition               : g2
   declared type             : Loop [Int]
   inferred type             : Bool
 full reduction              : 1. Applied : Loop a = a
                                  From    : (149,5)
                                  Step    : Int <- Loop Int
                                  Reason  : left to right application
                                  Amount  : 1 time
                               2. Applied : Loop [a] = Loop a
                                  From    : (148,5)
                                  Step    : Loop Int <- Loop [Int]
                                  Reason  : left to right application
                                  Amount  : 1 time
```

9.3 Quality of Error Reporting

In this section, we test the error messages against the manifesto by Yang et al. [39]. We explained this manifesto in section 3.3.1. We discuss each criterion and test how well our error messages do. We omit an explanation of the *unbiased* criteria as this is more a result of the type graph that Helium uses than the heuristic.

1. **Correct.** This criterion is linked to the performance of the type inference system. The system is built using the specifications as given by Vytiniotis et al. [36] which is proven correct. Apart from potential bugs present in the code, we may assume that the found erroneous constraint is correct. Furthermore, if the types in a given program are incorrect, the inference system will find an error. The reporting of the error in case of our heuristic always contributes to the problem if the problem is linked to a type family. All cases of the erroneous constraint require at least one of the types to be linked to a type family (using a reduction trace).

2. **Precise.** The error concerning this heuristic is shown using the erroneous constraint and its context. This context is often the type signature but may be any type annotation present in the code. This type annotation holds the type family that was the source of the error. The location of the type annotation is always given. Furthermore, the apartness hint mentions an instance from which a type family was not apart. In this case, we also provide the precise location of this instance to the programmer.

3. **Succint.** In our opinion, providing the types that don't match in the context of the definition they occur in is all relevant information. The full reduction that is given may help novice programmers to understand what happened exactly with the type family they specified. As mentioned, we only provide hints for the most deeply nested type families. These are the earliest errors and are one of the reasons the parent type families cannot reduce. The fact that the parent type family cannot reduce is not relevant to mention as the nested type family error should be solved first.

4. **A-mechanical and source-based.** The types in the erroneous constraint are fully substituted back to hold only type variables and types that are present in the source code. Therefore, the amount of a-mechanical information is limited. It is however possible that a type family is shown that is not present in the source code as it was partly reduced. In this case, we show what type family it was reduced from. The heuristic then requires that this type family is always present in the source code. We have, however, not been able to completely test this due to restrictions on features in the Helium compiler. The reduction trace shows constraints in case a type family is subjected to an injective adaptation. In this case, however, we feel that this constraint provides valuable information about the application of the type family.

5. **Comprehensive.** In case we obtain a type family with a nested type family as part of the erroneous constraint, there are two erroneous constraints. If we have constraint $F\ (G\ Int)\ Float \sim Int$ and $G\ Int$ cannot reduce, we get the following two constraints:

   (a) $F\ \beta_1\ Float \sim Int$

   (b) $G\ Int \sim \beta_1$

   In essence, these are two erroneous constraints that result from the same error, namely that $G\ Int$ does not reduce. Due to the choice of graph modifier (Section 9.2.4), we lose the second constraint after the heuristic blames the first one. We fixed this by considering nested type families during hint generation as explained in this chapter. Furthermore, the graph modifier may remove other erroneous constraints that occur from the type signature. We therefore cannot state that our heuristic supports fully comprehensive error reporting.

9.4 Interaction with other heuristics

The type family reduction heuristic is added to an environment in which other heuristics already compete to blame certain errors. We do not discuss these other heuristics in detail in this thesis but we discuss those with which this heuristic interacts. We mainly focus on those heuristics that may fire if this heuristic fails to do so or fire at the same time. In these cases, we elicited two situations in which this happens that we discuss here.

9.4.1 Type signature is too general

When we remove the type family reduction heuristic from the voting heuristic environment, type family related errors are caught by other heuristics. In case the erroneous constraint contains an unapplied type family, it is labeled as a *residual constraint*. In this case, one of two things may happen, based on whether the type family was partly applied or not or if it contains some type variables. Either the *type error is too general* heuristic fires or the constraint is caught by the filter heuristics implemented in Helium. We first present an example in which the type error is too general heuristic fires.

```
type family Perm a b
type instance Perm Float Int = Int
type instance Perm Int Float = Char
```

```
returnN :: Perm (Perm Int Float) Float
returnN = 'N'
```

Here, `Perm Int Float` is reducible, but the result family, `Perm Char Float`, is not. The error is as follows.

```
(262,1): Type signature is too general
  function                  : returnN
    declared type           : Perm (Perm Int Float) Float
    inferred type           : a
  hint                      : try removing the type signature
  full reduction            : 1. Applied : Perm Int Float = Char
                                 From    : (259,1)
                                 Step    : Perm Char Float <- Perm (Perm Int Float) Float
                                 Reason  : left to right application
                                 Amount  : 1 time
```

Notice that only the context information, namely the type signature, is shown. We thus lose some preciseness in the error, namely what the exact error was. However, we do have a trace! In some heuristics, the trace is also implemented as extra information, about which more in section 9.4.2.

In the other case, a similar error is returned but the constraint to blame is chosen by the *filter heuristics* implemented in Helium. In the case of the following program

```
type family Perm a b
type instance Perm Float Int = Int
type instance Perm Int Float = Char


returnN :: Perm (Perm Int Int) (Perm Float Float)
returnN = 'N'
```

the *participation ratio* filter heuristic immediately blames the correct constraint and returns the following error.

```
(262,1): Type signature is too general
  function                  : returnN
    declared type           : Perm (Perm Int Int) (Perm Float Float)
    inferred type           : Char
```

which in essence only shows the context in which the error resides. The error loses preciseness and a lot of helpful information surrounding the type family but it still points at roughly the right location. Furthermore, notice that in this case, no reduction trace is provided. Currently, this is only implemented for other *voting heuristics*.

In case a type family fully reduces but the resulting type takes part in a type error, we lose more information if the type family reduction heuristic is turned off. Consider the following example

```
type family Perm a b
type instance Perm Float Int = Int
type instance Perm Int Float = Char


returnN :: Perm Float Int
returnN = 'N'
```

where `Perm Float Int` reduces to `Int`. The error becomes a simple type error that does not mention the type family at all, like below.

```
(263,9): Type error in right hand side
  right-hand side           : 'N'
    type                    : Char
    does not match          : Int
```

However, if another voting heuristic were to fire upon this erroneous constraint, we may have another story.

9.4.2 Reduction trace in other heuristics

This other story is discussed in this section. We already saw an error message, resulting from another heuristic than the type family reduction heuristic, which showed a trace. In case the situation occurs that the type error reduction heuristic fails to fire for some reason, or that another heuristic fires that takes precedence, we would still like to show the potential reductions that took place. This provides more information to the programmer that may help solve the error. Say we adapted the Perm example program as follows.

```
type family Perm a b
type instance Perm Float Int = Int
type instance Perm Int Float = Float


returnN :: Perm (Perm Int Float) Int
returnN = 3.0
```

`Perm (Perm Int Float) Int` reduces to `Int` in two steps. The definition of `returnN` is of type `Float`. We thus obtain the erroneous constraint $Int \sim Float$. In such a situation, Helium defines the *sibling literals* heuristic that suggests changing the float literal to an int literal. Currently in Helium, we get the following error message.

```
(263,11): Type error in literal
  expression             : 3.0
    type                 : Float
    expected type        : Int
  probable fix           : use an int literal instead
  full reduction         : 1. Applied : Perm Float Int = Int
                              From    : (258,1)
                              Step    : Int <- Perm Float Int
                              Reason  : left to right application
                              Amount  : 1 time
                           2. Applied : Perm Int Float = Float
                              From    : (259,1)
                              Step    : Perm Float Int <- Perm (Perm Int Float) Int
                              Reason  : left to right application
                              Amount  : 1 time
```

We add the reduction trace to this error to let it make more sense. The trace makes it clear that the `Int` shown in the initial message was reduced from a type family in the type signature. Currently, we expanded the following heuristics to include the reduction trace:

- the type signature is too general heuristic,
- the sibling literal heuristic,
- the sibling functions heuristic,
- the application heuristic,
- the variable function heuristic and
- the tuple heuristic.

9.5 Conclusion

The type family reduction heuristic is a general heuristic that handles constraints that relate to erroneous type family reductions (or applications). To this end, the heuristic heavily relies on the reduction trace discussed in section 8. The trace allows us to provide more detail about how the types in the erroneous constraints were transformed during type inference. The quality of the given error message is quite high, however, we had to compromise on the comprehensiveness of the error reporting. We recognized that a fully reduced type family may trigger other heuristics in the heuristics environment of Helium. We, therefore, expanded those heuristics to also provide a trace if one were to be found in the given constraint. Because the heuristic relies so heavily on the reduction trace and we discussed its shortcomings, there are certainly situations in which the heuristic would fail to fire. In this case, other heuristics in the environment will take over. This results in error messages that are less precise but point at roughly the correct location. They may even show the reduction trace, as we saw in section 9.4.1.

Chapter 10

The Type Family Injectivity Heuristics

The fact that type families may be injective in Helium, means that the type system becomes more expressive. More programs may be type correct or can be made type correct by introducing some sort of injectivity. It also introduces more points of confusion for the programmer when type-checking a program fails. In this section, we discuss two new heuristics that focus on the injectivity of type families. The first, the *Inject Untouchable Heuristic*, signals when the programmer tries to inject an untouchable type variable. The second, the *Change Injectivity Annotation Heuristic*, tries to suggest adding or changing the injectivity annotation of a type family in case it may fix a certain type error. This section first lists a set of design requirements. Next, we discuss the implementation of both heuristics. After that, we discuss the quality of the error reporting that results from these heuristics and how they interact in the complete heuristic environment.

10.1 Design requirements

The heuristics described in this section are more specific than the type family reduction heuristic in the sense that they are built toward more specific situations. We may therefore see them as an extension of this heuristic. For example, if the injectivity annotation heuristic fails to fire, the reduction heuristic should back it up. The two heuristics explained in this chapter should be specific to their goal and only fire when their specific requirements are met. We, therefore, explain the goals of these heuristics separately before discussing their implementation and do not devise specific design requirements here.

10.2 Inject Untouchable Heuristic

In this section we discuss the *Inject Untouchable Heuristic* in detail. We discuss the reason why the heuristic was designed and its goal in designing an error message. Next, we describe its implementation in Helium.

10.2.1 Goal

When a type family is injective, new information may be found about its potentially previously unknown arguments. Those arguments are in the form of a type variable. In OutsideIn(X) and Rhodium, a type variable can be either *touchable* or *untouchable*. A touchable variable can be assigned a specific type during type inference. An untouchable variable cannot be assigned a specific type. This difference is introduced to be able to decide between whether a variable is quantified with an *exist* (∃) or a *forall* (∀) respectively. In the latter, untouchable, case we may not assume that the variable is any specific type because its enclosing type must be correct for any type that we may assign to the variable. In the version of Haskell currently implemented in Helium, a type signature or annotation is implicitly quantified with a forall. In that light, let us discuss the following example program.

```
type family B a = r | r -> a
type instance B Int = Float
type instance B Float = Int

f :: B a -> Int
f x = x
```

At first glance, one might expect that this function type checks. The type inference system finds that $B\ a \sim Int$ and, due to the injective properties of B, we may create $B\ Float \sim Int$ which in turn reduces to $Int \sim Int$. However, `a` is untouchable. As a result, the constraint $a \sim Float$ obtained during toplevel improvement (Section 7.5.5), becomes erroneous. For novice programmers, this might be confusing. Furthermore, this property of a type system is important to learn. To that end, we have designed this heuristic to catch and explain this error.

10.2.2 Implementation

In this section, we continue with the program introduced above. To catch the specific error for which this heuristic is designed, we need to annotate the constraint $a \sim Float$ with the information that it resulted from a toplevel improvement. To this end, we extend the *Property* data structure in *ConstraintInfo* with the following constructor

$$\begin{aligned}\text{Property } p ::= & \ \dots \\ & \mid \text{ResultOfInjectivity } \tau_1\ \tau_2\ c\end{aligned}$$

where τ_1 represents the type family, `B a` in our example, τ_2 the right-hand-side, `Int`, and c the complete constraint. We add the latter since it simplifies showing the constraint.

The construction of the heuristic is similar to the type family reduction heuristic in the sense that it seeks to blame a type signature or type annotation. We construct a potential trace from τ_1 and check whether the last type from the trace or τ_1 is in the type annotation. If so, we check whether the constraint info from the erroneous constraint contains a ResultOfInjectivity constructor. If it does, we continue to construct the hint. This hint tells the programmer that the program tried to inject an untouchable variable. For our example program, the hint looks as follows.

> could not assign "Float" to "a". "a" is quantified with an (implicit) forall and cannot be assigned any specific type

It hits the nail on the head. It tells us that the injection is erroneous because `a` cannot be assigned a type. After all, it is untouchable. On its own, the hint only provides information about a specific constraint and gives no context to where it occurs. This context is given in the final error message. Last but not least, notice that we provide the information that the forall may be implicit. We do this because a type signature does not need an explicit forall in Haskell. If it is not the case, all variables in the signature are implicitly quantified with a forall.

Last but not least, we discuss the complete error message that results from this heuristic. The heuristic adds the following constructor to the *properties* of the constraint.

$$\begin{aligned}\text{Property } p ::= & \ \dots \\ & \mid \text{InjectUntouchable } (Maybe\ \tau_1, \tau_2)\ \tau_3\end{aligned}$$

Here, τ_1 contains the potential last type obtained from the trace. τ_2 represents the family type, `B a` in our example and τ_3 contains the right hand side of the constraint, `Int` in our example. If the InjectUntouchable property is found in the constraint info of the constraint, the following error is built. We explain the structure of the error with the introduced example program.

```
(194,1): Could not reduce a type family in an explicitly typed binding further
  could not match          :
    type                   : B a
    with                   : Int
  in definition            : f
    declared type          : B a -> Int
    inferred type          : b   -> b
```

```
because                  : could not assign "Float" to "a". "a" is quantified with a
                           (implicit) forall and cannot be assigned any specific type
injectivity info         : From       : B a ~ Int
                           Got        : a ~ Float
```

Notice that the structure of the error is similar to the structure that the type family reduction heuristic induces. The reason for this is that the type of error is similar: we could not reduce a type family further. The thing that changes is the reason why, in this case, erroneous injectivity. Notice that the hint is now put into context. We now know where a is situated in the type signature. Next, the error adds *injectivity info* that tells the programmer from where the constraint $a \sim Float$ comes from. This completes the link from the *because hint* to the context provided by the error message.

10.2.3 Ambiguity check

In GHC, an *ambiguity check* is implemented which checks for every type signature or type annotation if it is ambiguous. Simply said, an ambiguous type has a very low chance that it will type check. The error caught with this heuristic is also caught by the ambiguity check in GHC. Helium currently does not implement the ambiguity check, which means that this error is currently only caught by the heuristic. However, even if the ambiguity check was implemented for Helium, this heuristic still has its use. In GHC the ambiguity check may be turned off through the *AllowAmbiguousTypes* language extension. This has its use in certain programs and it should also be an option in Helium. Therefore, we keep this heuristic for if the ambiguity check was turned off in some future version of Helium.

10.3 Change Injectivity Annotation heuristic

The change injectivity annotation heuristic is a bit different from the other heuristic discussed in this thesis in the sense that it focuses on adapting the type family declaration rather than blaming the type signature on a certain function. In this section, we explain its goals and implementation in Helium.

10.3.1 Goal

With the possibility to give a type family injective properties, these properties may be used to fix programs that were otherwise unfixable. Introducing an injectivity annotation to a type family declaration may help fix type errors while having little impact on other existing code that uses the family. An injectivity annotation only makes the type family more expressive, not less. The goal of this heuristic is to provide a possible injectivity annotation if that annotation fixes the erroneous constraint. Note that this is a suggestion and it is in no way certain whether it is what the programmer wants and needs. The context of the program is not known and therefore we cannot use it to improve the message. Let us clarify the goal by providing an example where the heuristic is applied.

```
type family Foo a where
  Foo Int = Bool
  Foo Char = Float
```

`Foo` is a very simple type family. The situation becomes interesting when we consider the following constraint: $Foo\ a \sim Float$. Currently, this constraint fails because the type family has no injectivity annotation. We may therefore in no way inject a. However, by making argument `a` injective, the constraint can be solved. $Foo\ a \sim Float$ becomes $Foo\ Char \sim Float$ which in turn is reduced to $Float \sim Float$.

10.3.2 Implementation

The heuristic only considers constraints of the form $F\ \bar{\sigma} \sim \tau$. $\bar{\sigma}$ can contain any combination of type variables or other types. The heuristic only fires if at least one possible annotation is found or if injectivity could help but the type family violates its properties. To find a possible annotation the heuristic performs the following steps:

1. We first check if τ unifies with any of the right-hand sides of the instances of type family F. If there is no right-hand side with which it unifies, we may stop, as no injectivity can be applied.

2. Next, the heuristic obtains type variables in $\bar{\sigma}$ that are touchable and *not* already injective. We call this set of variables $\bar{\epsilon}$. These variables may potentially be made injective to fix the erroneous constraint. If there are none, we stop as we are not able to suggest new injectivity annotations.

3. From this set of touchables $\bar{\epsilon}$, we compute the powerset without the empty case $\mathcal{P}_{-\emptyset}(\bar{\epsilon})$. For each set of touchables inside the powerset, $\bar{\epsilon}_{\mathcal{P}}$, we check if adding the declaration variables respective to their location in the injectivity annotation fixes the constraint. To do this, we first adapt the injective axiom of F to hold these new declaration variables. Next, we check if the new injectivity annotation passes the static checks for injective type families. If we don't do this, we run the risk of suggesting a new annotation that is then denied by the compiler during the static checks. If the static checks pass, we type check the erroneous constraint again using the new set of axioms. If that succeeds, we pass back $\bar{\epsilon}_{\mathcal{P}}$ unioned with the variables that were already injective in the axiom. This new set of variables will form one of the new annotations that will be shown in the hint of the error message.

The heuristic checks all $\bar{\epsilon}_{\mathcal{P}}$ in $\mathcal{P}_{-\emptyset}(\bar{\epsilon})$ and obtains all that succeed. If, after having checked all possible new annotations, some new annotations are found, these are listed inside a *possible fix* hint message. This message is structured as follows, using the example above.

> Add one of the following injectivity annotations to "Foo":
> 1: r $\rightarrow$ a

The only possible annotation that can be added is the one with the variable a. This is the only variable of `Foo`. If we add this annotation, the error is fixed.

If no new annotations were found, we know that there is no way that injectivity may help to solve the constraint. In this case, the heuristic looks at every entry in the powerset, $\mathcal{P}_{-\emptyset}(\bar{\epsilon})$, and obtains the variables that are inside every entry $\bar{\epsilon}_{\mathcal{P}}$. These variables can be assigned the label 'culprit'. Every time these variables are present, the injectivity checks fail. Knowing which variables these are, we may build a clear message to show to the programmer. To show this, we adapt `Foo` as shown below.

```
type family Foo a
type instance Foo Int = Float
type instance Foo Char = Float
```

The hint given is a *probable cause* hint and looks as follows.

> usage of type family "Foo" requires argument "a" to be injective, but "a" currently can't be

The first argument of `Foo` cannot be injected because it may assume two different types when the *rhs* of `Foo` is of type Float. The family, therefore, violates the injective property and as a result, cannot be made injective. The constraint $Foo\ a \sim Float$ cannot be fixed using injectivity of the type family.

The error message induced by the heuristic is similar to the message provided by the *type family reduction heuristic*. The only thing that changes is the hint provided. The error provides the exact type error in the context that it occurs in. Furthermore, it provides a reduction trace if one is present. The complete error message given for our example with `Foo` when it may be injective, is given below.

```
(240,8): Type family in variable was not fully reducible
  could not match            :
    type                     : Foo t
    with                     : Float
  in expression              : clsF
    declared type            : Bar a => Foo a
    inferred type            :          Float
  possible fix               : Add one of the following injectivity annotations to
                               "Foo":
                               1: r -> a
```

In case `Foo` cannot be injective in its (single) argument, the complete error looks as follows.

```
(240,8): Type family in variable was not fully reducible
  could not match          :
    type                   : Foo t
    with                   : Float
  in expression            : clsF
    declared type          : Bar a => Foo a
    inferred type          :          Float
  probable cause           : usage of type family "Foo" requires argument "a" to be
                             injective, but "a" currently can't be
```

Notice that the error looks similar with the only difference being that the hint has changed from a possible fix to a probable cause. In case the heuristic is not able to find a new annotation, it is not able to provide a fix. Instead, it calculates the cause of the error and shows it to the programmer.

10.4 Quality of Error Reporting

Because the error construction of both heuristics is similar to the error construction of the *type family reduction heuristic*, we do not discuss every criterion of the manifesto by Yang et al. [39]. Instead, we discuss two points of interest with respect to the *inject untouchable heuristic.*

10.4.1 Inject Untouchable Heuristic

The first criteria to discuss for the inject untouchable heuristic is *preciseness.* In case the type family in the type signature contains two type variables that must be injected for type inference to succeed, Helium currently only shows an error message for one. This violates the *preciseness* criterion that states that every conflicting location should be mentioned. Like the type family reduction heuristic, this heuristic removes the complete type signature from the type graph if it fires. As a result, other type variables that are erroneously injected, may not be considered. Again, this is not limited to the type of error that this heuristic considers. The removal of the type signature may impact whether another heuristic fires as well.

```
(194,1): Could not reduce a type family in an explicitly typed binding further
  could not match        :
    type                 : B a
    with                 : Int
  in definition          : f
    declared type        : B a -> Int
    inferred type        : b   -> b
  because                : could not assign "Float" to "a". "a" is quantified with a
                           (implicit) forall and cannot be assigned any specific type
  injectivity info       : From       : B a ~ Int
                           Got        : a ~ Float
```

Last but not least, the error message is a bit mechanical. The *injectivity info* part of the error shows a set of constraints, which is not source code based. Both constraints are generated during the type inference process and are therefore mechanical. We, however, feel that these specific constraints provide valuable information about what happened. Furthermore, they provide context for the hint that is given. Without the injectivity info, the types `Float` and `a` are introduced suddenly and without a link to the type error. We thus chose to include these constraints regardless of their impact on the *a-mechanical* criteria.

10.5 Interaction with other heuristics

The idea behind the heuristics discussed in this chapter is that they are targeted at more specific errors concerning type families. These heuristics focus on injectivity problems. In case they do not fire, but the error is type family related, we know that the error probably is not related to injectivity or can be fixed with it. In this case, the error is caught by the more general *type family reduction heuristic.* The *change injectivity annotation heuristic* only fires if the erroneous constraint is of the form $F\ \bar{\sigma} \sim \tau$. If the given constraint is of that form and the heuristic does not fire, the probability that the type family reduction heuristic fires is very high. It has a case for a constraint of that form and only does not fire if the type family in the constraint is not part of the type signature. We may therefore say that the type family reduction heuristic acts as a fallback heuristic for the change injectivity annotation heuristic.

The *inject untouchable heuristic* does not have the type family reduction heuristic as a fallback heuristic. It reacts on a constraint of the form $tv \sim \tau$ where tv is a type variable. This constraint contains a specific entry in its constraint info that causes the heuristic to fire. If its constraint info misses this entry, it is hard to determine what other heuristic may fire. We suspect this will often be the *type error is too general* heuristic. Especially if tv is untouchable. Then, the type variable is probably part of a type signature to which the programmer tries to assign a specific type. This is not possible, as the variable is quantified with a forall.

10.6 Conclusion

In this chapter, we discussed two more specific heuristics that are built to find type errors related to the injectivity of type families. The inject untouchable heuristic is built to catch a type error for which GHC uses the *ambiguity check*. As this is currently not implemented in Helium, we built this fallback heuristic to catch the error and provide helpful information. The heuristic focuses on the situation where the programmer tries to inject a type into an untouchable heuristic. The resulting error message violates the manifesto by Yang et al. [39] in the sense that the provided injectivity info is a bit mechanical. We, however, feel that this is needed to provide the necessary information to understand the error.

The second heuristic that was discussed, is the change injectivity annotation heuristic. This heuristic diverts its focus from the type signature and instead tries to find if changing the injectivity annotation of a type family might fix the erroneous constraint. The heuristic provides a link between non-injective and injective type families. It is also able to suggest injectivity if the type family does not yet have an annotation. The heuristic only fires if there is a possibility that injectivity may help by checking if the *rhs* of the instance unifies with the *rhs* of the erroneous constraint. Furthermore, if no working injectivity annotation may be found, the heuristic states this as a possible cause. It provides the arguments that currently violate the property. If the heuristic does not fire at all, the type family reduction heuristic is its most probable fallback due to the type of constraints both heuristics fire on.

Chapter 11

Comparison of errors between GHC and Helium

In this chapter, we compare the type error messages given in GHC with the type errors we designed in Helium. For example, we discuss the differences and their effect on the understandability of the type error that induced it. The examples remain simple, as Helium only supports type families in the simplest sense.

11.1 Example 1: Wut

The first example that we discuss, is a simple one. We introduce the following example program.

```haskell
type family Foo a
type instance Foo Int = Float

wut :: Foo Float
wut = 6
```

Because `Foo Float` is not reducible, we obtain a type error because the constraint *Foo Float* ∼ *Int* is erroneous. This is the clearest message that the compiler can give. However, GHC gives the following error message.

```
main.hs:179:7: error:
• No instance for (Num (Foo Float)) arising from the literal '6'
• In the expression: 6
    In an equation for 'wut': wut = 6
    |
179 | wut = 6
    |
```

The error blames the fact that there is no instance `Num` for the type `Foo Float`. This error arises from the fact that the compiler inferred the type `Integer` from the value `6`. This type has an instance of `Num` and as a result, the compiler expects that one exists for `Foo Float` as well. Here, we see that GHC heavily treats type families as normal, total, types for which instances can be written. In Helium, we blame type family errors on type signatures and type annotations. In case a type family is not reducible, Helium notices this and blames it. Helium, therefore, provides the following error message.

```
(274,1): Type family in explicitly typed binding was not fully reducible
could not match           :
  type                    : Foo Float
  with                    : Int
in definition             : wut
  declared type           : Foo Float
  inferred type           : Int
probable cause            : "Foo Float" is not reducible. No matching instance was found
```

The error mentions the type mismatch in its context as we explained when we discussed the heuristic. Note that the error shows the exact probable cause, namely that `Foo Float` is not reducible. Overall, this error provides more understanding of the actual error than the error that GHC gives. The error from GHC is a result of the fact that it supports *overloaded numerals*. As a result, the compiler thinks that `Foo Float` should have a numerical instance. Helium does not support overloaded numerals (in this branch, at least) and therefore does not consider this. In case Helium would support this, we still think that Helium gives the same type error. Two error paths would emerge from type inference and the path of our current error would probably be the longest. Therefore, this error takes precedence and is shown.

11.2 Example 2: Loop

In case a type family is not reducible because the apartness check for closed type families fails, GHC does not mention that this is actually the cause. To show this, we use the following, recurring, example.

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

loop :: Loop [c]
loop = 'X'
```

After `Loop [c]` reduces to `Loop c` using the first instance, a programmer inexperienced with type families might think that the next step will be the application of the second instance to form `c`. This, however, does not happen because `Loop c` is not apart from `Loop [c]`. `c` is not apart from `[a]` because `c` can be instantiated as `[a]` like we explained in section 9.2.2. GHC provides the following error.

```
main.hs:206:8: error:
    • Couldn't match type ‘Loop c’ with ‘Char’
      Expected type: Loop [c]
        Actual type: Char
    • In the expression: 'X'
      In an equation for ‘loop’: loop = 'X'
    • Relevant bindings include
        loop :: Loop [c] (bound at main.hs:206:1)
    |
206 | loop = 'X'
    |
```

The correct error is given but no context as to why `Loop c` is not reduced is given. Our error in Helium does provide this context in form of the *apartness hint* that we discussed in section 9.2.2. The error looks as follows.

```
(281,1): Type family in explicitly typed binding was not fully reducible
  could not match              :
    type                       : Loop c
      reduced from             : Loop [c]
    with                       : Char
  in definition                : loop
    declared type              : Loop [c]
    inferred type              : Char
  probable cause               : type "Loop c" is not apart from instance "Loop [a]" at (278,5)
  full reduction               : 1. Applied : Loop [a] = Loop a
                                    From    : (278,5)
                                    Step    : Loop c <- Loop [c]
                                    Reason  : left to right application
                                    Amount  : 1 time
```

Notice the *probable cause* hint given that tells the programmer that apartness was to blame for the failed application of `Loop c`. Furthermore, our error provides the reduction trace that tells the programmer that the application of the first instance did succeed! This narrows down the field in which the programmer has to search for the error. The first instance is at least not to blame!

11.3 Example 3: Injectivity

Using the next example, we showcase the differences in type error messages between GHC and Helium that have to do with injective type families. We first compare error messages related to the error that the *inject untouchable heuristic* catches. We use the same program that we used to showcase the heuristic in section 10.2.

```
type family B a = r | r -> a
type instance B Int = Float
type instance B Float = Int

f :: B a -> Int
f x = x
```

Remember that `a` in `B a` cannot be injected because it is an untouchable variable. As a result, `B a` cannot be reduced any further. GHC provides the following type error message.

```
main.hs:360:7: error:
    • Couldn't match expected type ‘Int’ with actual type ‘B a’
    • In the expression: x
      In an equation for ‘f’: f x = x
    • Relevant bindings include
        x :: B a (bound at main.hs:360:3)
        f :: B a -> Int (bound at main.hs:360:1)
    |
360 | f x = x
    |       ^
```

The correct type error is provided without any explanation. Notice that the error locates the `x` in the body of the function as the source of the error. It is, however, not certain that that is the location to blame. It may as well be the `x` in the argument of the function. Helium provides an error message with more context information, like below.

```
(301,1): Could not reduce a type family in an explicitly typed binding further
could not match            :
  type                     : B a
  with                     : Int
in definition              : f
  declared type            : B a -> Int
  inferred type            : b   -> b
because                    : could not assign "Float" to "a". "a" is quantified with a
                             (implicit) forall and cannot be assigned any specific type
injectivity info           : From       : B a ~ Int
                             Got        : a ~ Float
```

We discussed the contents of the message in section 10.2. In this section, we focus on the differences with the message obtained from GHC. First, notice that the error given by Helium keeps the exact location of the error more general. The declared type and inferred type of the function are given and it is up to the programmer to locate the error. We do this to prevent leading the programmer to a piece of code that has nothing to do with the error or is not the most specific location. Secondly, the error explains the error and what the effect of injectivity was during type inference. From this, the programmer can learn how injective properties of a constraint may introduce new constraints to make the type inference system more expressive. The error in GHC does not provide such information.

11.4 Example 4: Injectivity annotations

Last but not least, we discuss the error messages that GHC and Helium give in situations where adding or adapting injectivity annotations may fix the error. To this end, we introduce the type family `Inj`.

```
type family Inj a b where
type instance Inj Int Float = Char
type instance Inj Float Float = Int
```

Notice that the type family has no injectivity annotation and is therefore considered not injective by both GHC and Helium. Next, let us consider the constraint `Inj a b ~ Char`. In this case the constraint would be solved if both `a` and `b` were injective. Notice that this does not violate the requirements for an injective type family. GHC gives us the following error.

```
main.hs:373:5: error:
    • Couldn't match expected type ‘Char’ with actual type ‘Inj a0 b0’
      The type variables ‘a0’, ‘b0’ are ambiguous
    • In the expression: inj
      In an equation for ‘x’: x = inj
    |
373 | x = inj
    |     ^^^
```

The error mentions the correct erroneous constraint. Next, it mentions that ‘a0’ and ‘b0’ are ambiguous. First of all, these type variables are introduced by the compiler during the type inference process and are not present in the source code. For a programmer that is less experienced with what happens during type inference, it is unclear what the zeroes mean. Last but not least, the error message mentions that these variables are ambiguous, meaning that they are open to more than one interpretation. This is true. We can never assign one exact type to these variables. For a novice programmer, however, this notion can be confusing and difficult to understand. Why is this the case? In Helium, we remove the mention of ambiguity altogether and focus on the real problem at hand: injective usage of the type family. The Helium compiler gives the following error message.

```
(295,5): Type family in variable was not fully reducible
could not match          :
  type                   : Inj a b
  with                   : Char
in expression            : inj
  declared type          : InjC a => Inj a b
  inferred type          :           Char
possible fix             : Add one of the following injectivity annotations to "Inj":
                           1: r -> a b
```

The type mismatch is the same as in the message from GHC. The difference between the messages lies in locating the reason for the error. The message from Helium can blame the lack of injectivity as the cause. Compared to the message of Helium, which is more general, the message given by Helium is much better. Helium even provides a possible fix that helps to solve the error. As we have shown in section 10.3, Helium also blames the lack of an injectivity annotation when one cannot be added. In this case, the compiler tells the programmer that the injective usage of the type family cannot be solved by changing or adding an injectivity annotation.

Chapter 12

Conclusion

In chapter 6, we posed the following main research question:

> *How can causes of type errors involving Type Families be discovered and explained using heuristics?*

We have shown that it is possible to discover and explain type errors involving Type Families using the techniques that Helium provides. The provided error paths from the type graphs are correct and provide enough information to find out what went wrong. We added several heuristics that were specifically designed to deal with the addition of Type Families. These heuristics add a new layer to the heuristic environment of Helium using Rhodium. The errors related to Type Families were already caught by the type inference system itself and we feel that the newly, specialized, heuristics allow the compiler to report error messages of higher quality. They provide the programmer with hints toward a fix and give the incentive to learn what happens with Type Families during type inference.

This incentive is enlarged by adding the Reduction Trace to error reporting, as an answer to the second sub-question posed in chapter 6. By providing each reduction step of a Type Family in detail, the programmer is able to see what effect certain properties, like injectivity, have on them during applications. Obtaining this information from the type inference process is hard because type families are not necessarily applied in order of nested first, parent after. Therefore, the currently implemented reduction trace is not perfect. In case the compiler fails to construct one, it may fall back onto a less detailed error message. We feel, however, that this less detailed error message still is an improvement when comparing it to the default in Helium.

Last but not least, we answer the first sub-question as posed in chapter 6. We found that it was relatively easy to implement all types of type families in Helium. An associated type synonym is simply an open type family with some extra checks that relate to its parent type class. After these checks are performed, it is simply considered an open type family. Closed type families required adaptations to the type inference process. By adding the *Axiom ClosedGroup*, we were able to distinguish closed type families from open ones. In the case of type error messages, the *apartness hint* was specifically designed for closed type families. This check is only performed for this type of type family and is one that may be confusing. Injective type families required a whole different set of static checks to be implemented. During type inference, an extra step was implemented to find new information on type family arguments. This, however, does in no way clash with the normal, left to right, application of type families. With respect to type error messages, separate heuristics were designed that overrule the standard type family heuristic. We feel that this was the right way to go because injectivity makes a type family more expressive. In case the more specific injectivity heuristic fires, it means that this expressiveness is needed and we should show it to the programmer.

All in all, the current implementation of Type Families in Helium, including the heuristics designed for it, greatly improves the understandability of type error messages related to Type Families. As a last note, we would like to mention that usage of Type Families in Helium is somewhat limited due to the absence of language extensions like Data Kinds. Implementing such language features may result in revisiting the type family heuristics as they are now.

Chapter 13

Future Work

13.1 Improve the Reduction trace

One conclusion of this thesis is that the current implementation of the reduction trace is not perfect. This has to do with the non-linear nature of the type inference system. We suggest researching other approaches to obtain reduction steps and the trace. The type graph that Rhodium uses may, for example, be reused to this end. The graph saves all constraints in the constraint system and their relations. To this end, it also knows the parent and child constraints of every constraint. It may therefore be a good idea to build an algorithm that walks the type graph to find a type family reduction trace.

13.2 Design a more specific graph modifier

The heuristics currently implemented for type families in Helium use a somewhat drastic graph modifier. This modifier removes the complete type signature in which an erroneous type family resides. We suggest that some way should be found to make this graph modifier more specific. We experimented with only replacing the type that is the culprit of the error with the type that it should be. This has disadvantages. For example, this approach may introduce new errors that were not there before. The new graph modifier should be a more specific version of the current one and should not introduce artifacts.

13.3 Implement supporting language constructs for Type Families

The example programs given in this thesis are all using type families that are quite simple. The reason for this is that Helium does not support many ‘supporting’ language constructs for type families. When looking at GHC, we noticed that type families are often used in combination with language extensions like ‘DataKinds’ and ‘UndecidableInstances’. DataKinds introduces a richer type system in which data structures are lifted to kinds and their constructors to types. UndecidableInstances removes some of the restrictions for type families with the risk of a looping type inference system. Both extensions, and there are more, expand the power of type families greatly. The example type family we introduced in section 2.6, cannot be implemented in Helium because type family ‘Append’ requires the DataKinds extension. We suggest researching how to implement these expanding extensions while preserving the type error diagnosis quality that Helium now boasts.

13.4 Merge type family branch with other branches in the Helium repository

The Helium compiler has been, and is, a home for many great pieces of research. Sadly, we noticed that all those researches are currently divided over many branches in the GitHub repository. After some experimenting, we noticed that merging all this research will be quite a job. For example, a new code generation backend has been designed that supports typing [7]. The research in this thesis only implemented type families up to the type inference phase in Helium. As a result, the new code generation backend must be adapted to implement type families as well. Apart from this example, there are more subtle difficulties that make merging all this research difficult. We, therefore, suggest allocating some time and effort to do this.

13.5 Impredicative types

Rhodium allows us to implement a type system analogous to the one in GHC because both type inference systems are the same logically. For GHC, research has been conducted on how to implement impredicative types, types that allow universal quantifiers to appear anywhere in a type. For Helium, the next step may be to implement impredicative types as well and to conduct research on how this affects the current work done on type error diagnosis.

13.6 Other language extensions

Apart from the language extensions we mentioned in section 13.3, there are many others. The X in Rhodium allows us to implement those. We may research developing a general framework for those language extensions and how these extensions affect type error diagnosis. Furthermore, we may look into designing heuristics for those extensions. It would be interesting to see if certain heuristics could be turned off in case certain language extensions are not used. Notice that this could also be the case for the type families implemented in this thesis.

Bibliography

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 31–41, 1993.

[2] K. L. Bernstein and E. W. Stark. Debugging type errors (full version). *Unpublished Technical Report, Computer Science Dept. State University of New York at Stony Brook*, 1995.

[3] J. Burgers. Type error diagnosis for outsidein(x) in helium. Master's thesis, Utrecht University, 6 2019.

[4] J. Burgers, J. Hage, and A. Serrano. Heuristics-based type error diagnosis for haskell: The case of gadts and local reasoning. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, pages 33–43, 2020.

[5] M. M. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. *ACM SIGPLAN Notices*, 40(9):241–253, 2005.

[6] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[7] I. G. de Wolff. The helium haskell compiler and its new llvm backend, 2019.

[8] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. *ACM SIGPLAN Notices*, 49(1):671–683, 2014.

[9] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen. Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27(1):65–100, 2017.

[10] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

[11] J. Hage. Domain specific type error diagnosis (domsted), 2014.

[12] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In *Symposium on Implementation and Application of Functional Languages*, pages 199–216. Springer, 2006.

[13] B. Heeren and J. Hage. Type class directives. In *International Workshop on Practical Aspects of Declarative Languages*, pages 253–267. Springer, 2005.

[14] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. *ACM SIGPLAN Notices*, 38(9):3–13, 2003.

[15] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71, 2003.

[16] B. J. Heeren. *Top quality type error messages*. Utrecht University, 2005.

[17] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[18] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[19] G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 44–57, 1986.

[20] M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, pages 230–244. Springer, 2000.

[21] Y. Jun, G. Michaelson, and P. Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.

[22] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):707–723, 1998.

[23] D. Leijen. LVM, the lazy virtual machine. Technical report, Citeseer, 2002.

[24] B. J. McAdam. Generalising techniques for type debugging. In *Scottish Functional Programming Workshop*, volume 1, pages 50–58. Citeseer, 1999.

[25] J. G. Morris and R. A. Eisenberg. Constrained type families. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

[26] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. *ACM SIGPLAN Notices*, 41(9):50–61, 2006.

[27] J. A. Robinson. Computational logic: The unification computation. *Machine intelligence*, 6:63–72, 1971.

[28] T. Schilling. Constraint-free type error slicing. In *International Symposium on Trends in Functional Programming*, pages 1–16. Springer, 2011.

[29] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, 2008.

[30] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, 2008.

[31] A. Serrano. Cobalt. https://github.com/serras/cobalt, 2015.

[32] A. Serrano. *Type Error Customization for Embedded Domain-Specific Languages*. PhD thesis, Utrecht University, 2018.

[33] A. Serrano and J. Hage. A compiler architecture for domain-specific type error diagnosis. *Open Computer Science*, 9(1):33–51, 2019.

[34] J. Stolarek, S. Peyton Jones, and R. A. Eisenberg. Injective type families for haskell. *ACM SIGPLAN Notices*, 50(12):118–128, 2015.

[35] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 72–83, 2003.

[36] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. Outsidein (x) modular type inference with local assumptions. *Journal of functional programming*, 21(4-5):333–412, 2011.

[37] M. Wand. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 38–43, 1986.

[38] J. Yang. Explaining type errors by finding the source of a type conflict. In *Scottish Functional Programming Workshop*, volume 1, pages 59–67. Citeseer, 1999.

[39] J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In *In Proceedings of 12th International Workshop on Implementation of Functional Languages*. Citeseer, 2000.

[40] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 569–581, 2014.

Appendix A

Setting up Helium

For those readers who want to try our work themselves, we provide the steps to build and run the `thesis-typefam` branch of `Helium`. Helium relies on some other repositories, which need to be set up first. All repositories are part of the Helium4Haskell organisation on Github. Every clone should be positioned in the same parent folder.

1. Clone the Top library (`https://github.com/Helium4Haskell/Top`). Ensure that in `Top.cabal`, the line `cabal-version: >=2` is removed. Otherwise, stack won't build.
2. Clone the lvm library (`https://github.com/Helium4Haskell/lvm`) and checkout the branch `thesis-typefam-compatible`. This branch is a modified version of an older commit (`fc80b24`) that changes monoid instances to use the new semigroup instances.
3. Clone the Rhodium library (`https://github.com/Helium4Haskell/rhodium`).
4. Install the specific `uuagc` library. Instructions below:
   (a) git clone `https://github.com/noughtmare/uuagc.git`
   (b) cd uuagc
   (c) git checkout mirage
   (d) git pull
   (e) cd uuagc/trunk
   (f) cabal v1-install
5. Clone Helium itself from `https://github.com/Helium4Haskell/helium` and checkout the branch `thesis-typefam`.

At this point, you have everything you need to run the compiler. In `lib/Test.hs`, you can find programs and write your own. To build and type check your type family programs, perform the following steps:

1. `cd src/` and run `make`. This generates the needed Haskell files from the .ag files in the code. After you did this, you only have to repeat it if you adapt .ag files.
2. `cd ../` and run the following large command:

```
stack run helium -j12 -- -b --stop-after-type-inferencing --gadts
--showtrace --outsideinx --basepath=./lib/ ./lib/Test.hs
```

This command requires some explaining. `-j12` tells the compiler to run in multithreaded mode using 12 cores. You may adapt it to suit your machine. `-b` tells the compiler to only build the program, not run it. `--stop-after-type-inferencing` lets the compiler stop after the type checking and inference phase. Because type families are not incorporated in code generation yet, it is best to stop here. `--gadts` allows you to write GADTs in your code, neat right? `--showtrace` is the argument that lets the compiler show the full trace in an error message that contains it. Removing this argument results in removing the reduction trace from the error messages. `--outsideinx` tells the compiler that Rhodium should be used during type inference. Last but not least, we specify `./lib/` as the basepath for all the files that compilation needs. Have fun with Helium!