MEng Individual Project

Imperial College London

Department of Computing

Total Type Classes: Improving the ergonomics of type-level programming in Haskell

Author:
Robert Weingart

Supervisor:
Nicolas Wu

Second Marker:
Cristian Cadar

June 19, 2024

Abstract

The Haskell programming language is a powerful tool for type-level programming due to its powerful and feature-rich type system. While it lacks dependent types, which would allow arbitrary information about a program to be directly encoded in types, modern Haskell offers a wide range of type system features which can work around this omission. However, these approaches can be unintuitive and cumbersome, as they reuse Haskell features whose syntax and semantics were initially designed for a different use case.

We identify one such issue, namely that the intuitive interpretation of type classes as constraining the possible values of a type parameter clashes with their use as functions from types to terms. To alleviate this, we introduce the notion of *total type classes*, which are those whose instances cover all possible types inhabiting a particular kind, which is possible whenever that kind is a promoted algebraic data type.

Our technical contribution is a compiler plugin which allows methods of total type classes to be called even when the relevant type class constraint is not given in the enclosing function's type signature, which enables the programmer to treat these methods like ordinary functions. To achieve this functionality, the plugin records the missing constraint during typechecking and then modifies the typechecked program, rewriting it as though the programmer had included the constraints to begin with. The plugin also provides a checker which identifies total type classes.

Our implementation achieves its primary goal, accepting a large number of total type classes and accurately rewriting a wide range of programs. We also identify a number of ways to extend this functionality to cover more use cases, which we leave for future work.

Acknowledgements

I am deeply grateful for all the help and encouragement that my supervisor, Nicolas Wu, has given me throughout this project. He gave me both the confidence and the freedom I needed to make this project something I could be proud of.

I also want to thank my parents for all the support they have given me over the years, which enabled me to be the person I am today.

Finally, I express my utmost gratitude to my partner Alex. During the greatest challenges of my life, this project included, I know I can always rely on their kind words and unending support, without which I doubt I could ever have made it this far.

Contents

Chapter 1

Introduction

Haskell is a programming language known for its powerful and expressive type system. However, the demand for more advanced type-level features has grown over time, as programmers find new ways to increase the safety of programs at compile time. The most popular Haskell implementation, the Glasgow Haskell Compiler (GHC), provides a number of unique features that increase the expressivity of Haskell's type system beyond the standard. The conjunction of these features allows programmers to implement design patterns similar to those which would be available in a language that supports full dependent typing.

However, these workarounds are not always ergonomic. One example is the way in which type classes interact with these new features. In standard Haskell, type classes are a crucial feature for abstracting over properties of a type, creating a standard interface for a type to conform to and then instantiating it for particular types. In the context of type-level programming, type classes are needed to control runtime behaviour using the rich information that is now present in types. Thus, they take on a role more similar to term-level functions, performing complex computations that transfer data from the type level to the term level. A unique product of this use case are type classes which have an instance for every type of a given kind, similarly to functions which are defined for every possible value of a given type.

However, this analogy with functions is limited because the compiler does not treat these *total type classes* any differently from others. For instance, any function which uses such a type class must mention a constraint as part of its type, e.g. writing `C a =>` at the start of the function signature, to express that requirement that `a` has an instance of class `C`, even though in this case the programmer knows that an instance will always exist. This creates a mismatch between the intuitive meaning of the code and the way that the programmer must write it to satisfy the compiler. This project rectifies this by introducing special compilation behaviour for such type classes, allowing a constraint to be omitted when all possible instances are known to exist.

1.1 Contributions

Our main contribution is a GHC plugin which provides the following features:

- The definition of a type class and its instances can be checked to verify that the class is total, in the sense that every possible argument matches an instance.
- When the absence of a given constraint for such a class would otherwise cause the compilation to fail, the code will instead be rewritten during compilation, as though the user had provided the constraint.

Our code can be found at `https://gitlab.doc.ic.ac.uk/raw20/fyp-plugin`

1.2 Ethical Discussion

No ethical issues arise from the contents of this report or the functionality of our implementation. All writing and code are original, except for the contents of `src/TotalClassPlugin/GHCUtils.hs`, which are copied with minor modifications from the source code of the Glasgow Haskell Compiler, which is permitted by that project's license [GHC Team, 2024].

Chapter 2

Background

We present an overview of some important concepts in type theory that programmers may seek to achieve in Haskell (Section 2.1). We then introduce a number of existing GHC features that augment the type system (Section 2.2) before presenting an example which showcases their use as well as the problem that our implementation solves (Section 2.3). Finally, we examine how GHC can be extended using plugins, which forms the basis of our implementation (Section 2.4)

2.1 Type theory

2.1.1 Parametric and ad-hoc polymorphism

We distinguish between *parametric polymorphism* and *ad-hoc* polymorphism [Strachey, 2000]. Parametric polymorphism means that the function's behaviour is independent of the type in question. One example would be standard list operations like `length` and `map`, having type signatures:

```
length :: [a] -> Int
map :: (a -> b) -> [a] -> [b]
```

These functions can be instantiated for any types `a` and `b` without changing the implementation. On the other hand, ad-hoc polymorphism allows for implementations to vary depending on the type at which a function is instantiated. Examples include numeric operators like addition, was well as conversion to a string. These operations should be "overloaded" for multiple different types, e.g.

```
(+) :: Int -> Int -> Int
(+) :: Double -> Double -> Double
show :: Int -> String
show :: Bool -> String
```

but we cannot derive the above functions from a single, generic implementation of `(+)` and `show`.

The Hindley-Milner type system, on which Haskell is indirectly based, supports parametric polymorphism well, but requires complex and often unsatisfactory workarounds for ad-hoc polymorphism [Strachey, 2000]. To combat this, Haskell introduced *type classes* as a standardised way of implementing ad-hoc polymorphism. Their design will be explored in detail in the next section.

2.1.2 Dependent type theory

Dependent types are types which can mention terms. For instance, a dependently typed programming language may have a type such as $\mathbb{N}^3$, containing 3-typles of natural numbers, and a function type such as $\Pi(n : \mathbb{N}) \to \mathbb{N}^n$, which accepts a natural number n as an argument and produces a tuple of length n. Using dependent types, a programmer can include arbitrary information at the type-level, potentially allowing for safer programs with very expressive types.

Currently, Haskell does not support dependent types, although their implementation has been suggested in the past [Eisenberg, 2017]. Nevertheless, modern Haskell has a number of features which can be used to simulate dependently-typed programming [Lindley and McBride, 2013]. We will explore some of these in the next section.

2.2 Prior development of the language

2.2.1 GHC

While there are a number of implementations of the standard described by the Haskell Report, this project focuses on the most popular one, GHC [Hudak et al., 2007]. This compiler includes a number of non-standard advanced features, including the type-level programming features at the heart of this project. GHC compiles Haskell into an intermediate language called `Core` [Marlow and Peyton Jones, 2012]. This *desugaring* step transforms the vast syntactic complexity of Haskell into a much simpler language, making its study and verification easier. Notably, `Core` is a strongly typed language. This provides safety during the design of new Haskell features: by describing a feature in terms of its translation into `Core`, its impact on the semantics of the language (and the type-safety thereof) can be evaluated. Most of the features discussed in this report were introduced in this way.

2.2.2 Type classes

Type classes are Haskell's distinctive approach to achieving ad-hoc polymorphism [Wadler and Blott, 1989]. Their initial purpose was to create a unified system for implementing numeric operations as well as equality, which should be polymorphic, but cannot be implemented parametrically. Type classes allow the programmer to constrain the generic parameters of a function, requiring an otherwise arbitrary type to have a certain property. This requirement can be satisfied by providing an instance of the class for a particular type, with an implementation specific to that type.

An example of a type class is the built-in class `Show`, which represents conversion to a string. When a programmer defines a data type `T` for which they want to provide a string conversion, they can define an *instance* `Show T` by implementing the type class method `show :: T -> String`. Other code can then use the method `show` on any type which implements `Show` (i.e. for which an instance exists), including `T`. An example of usage is:

```
show2 :: Show a => a -> String
show2 x = show x ++ " " ++ show x
```

This function accepts a value of any type `a`, given an instance `Show a` which implements conversion from `a` to `String`. It returns a string consisting of two copies of the string representation of the input value, separated by a space. To do so, it uses the function `show` whose existence is guaranteed by the `Show` constraint.

Terminology

For clarity, we make note of some terminology used for type classes. The contstruct `Show a` in the code above is called a *constraint*. When a constraint appears in a function, it is available in the body of the function as a *given constraint*; when the function is called, it gives rise to a *wanted constraint* instantiated according to the function's arguments (for instance, in the case of `show2 True` the wanted constraint will be `Show Bool`). The term used to solve a constraint (containing the application of the corresponding instance function, which will be a normal term in `Core`) is referred to as `evidence`. An instance declaration can itself have other constraints as preconditions:

```
instance Show a => Show [a] where ...
```

The constraint being provided (`Show [a]`) is called the *head* of the instance, while the constraints required (`Show a`) form the *context* [GHC Team, 2023, Section 6.8.8]. When GHC tries to solve a wanted constraint, it tries to match it with some instance head or given constraint. If an instance head matches, the constraints in that instance's context become new wanted constraints. See Section 5.1.4 for a more detailed discussion of how the constraint solver is implemented.

Implementation

The properties of type classes are formalised and implemented via their translation into `Core` [Hall et al., 1996]. Each type class is transformed into a polymorphic type of *dictionaries*. A dictionary for a class `C` is simply a tuple whose members are implementations of the methods. An instance with head `C a` becomes a function returning a dictionary, with arguments corresponding to the

instance's context. In the type of a function definition, the constraint `C a =>` becomes an ordinary argument `C a ->` whose type is the dictionary type. When such a function is called, the instance function is used to create a dictionary which is passed to the function.

The history of type classes gives rise to two different mental models of their operation. On one hand, we can think of a constraint like `Show` as a predicate on types. That is to say, `show :: Show a => a -> String` is a function which takes one parameter, of type `a`, provided that type `a` satisfies the property `Show`. The type `a` is then an instance of `Show`. On the other hand, we may think of the constraint `Show a` as a type, whose values are instances created from the instance definitions written by the programmer. In this model, `show :: Show a => a -> String` really takes two parameters: the instance satisfying the constraint `Show a` and the value of type `a`. As we saw, the latter is how type classes are actually implemented in GHC, whereas the former is simpler and often easier to explain. This generally does not affect the programmer's understanding of code, as the two models are essentially equivalent: By default, a particular constraint such as `Show String` can only have one value, since a program cannot contain multiple instances with overlapping heads. Thus, the information that `Show String` has a solution is (type-theoretically) equivalent to the solution itself, since it is unique. The fact that the solution is actually a piece of data that may need to be passed to the function at runtime is essentially an implementation detail.

In modern Haskell, type classes have many more uses than their initial purpose. For instance, they were extended with *functional dependencies*, which allow the programmer to write multi-parameter type classes where the choice of one parameter determines another [Jones, 2000]. Initially intended to make type inference for multi-parameter type classes feasible, they soon became an early approach to type-level computation [Hudak et al., 2007]. This project focuses on yet another use of type classes: to allow terms to depend on the complex type information that modern Haskell can encode. Indeed, we can think of type classes as functions from arbitrary type-level data to arbitrary term-level data. Unfortunately, in this situation, the equivalence between the two mental models we discussed is broken. Resolving this is the main goal of this project.

Kinds of classes

The GHC extension `ConstraintKinds` [GHC Team, 2023, Section 6.10.3] introduces the special kind `Constraint`, which is the type of constraints like `Show String`. Using this extension, classes and instances can be used in conjunction with type synonyms and and type signatures. For example, `Show` can be given the type `Type -> Constraint`, while `Functor` has type `(Type -> Type) -> Constraint`. Considering how classes are implemented via dictionaries, one can think of `Constraint` as the type of types of dictionaries, mirroring how `Type` is the type of ordinary types.

2.2.3 Data Kinds

The `DataKinds` extension is a relatively recent addition to GHC, with the goal of allowing for more expressive type-level programming in Haskell [Yorgey et al., 2012]. Suppose we have a simple data type, such as

```
data Nat = Z | S Nat
```

Here, Z represents zero, and the constructor S is the successor function which maps each natural number to the next one, so that S Z represents 1, S (S Z) is 2, and so on. Suppose we want to include information of type `Nat` in the type of another value. Since Haskell is not a dependently typed programming language, we cannot directly reference a term in a type. Without `DataKinds`, we could recreate the `Nat` type at the type level by defining:

```
data Zero

data Succ a
```

Note that these datatypes have no constructors, meaning that they do not contain any values; rather, we only use them to encode type-level information. However, the kinds of these types are simply `Zero :: Type` and `Succ :: Type -> Type`, meaning that we can write nonsensical types

such as `Succ Bool`. There is no way to define a generic type whose type parameter must be from a restricted set of types, the same way we restrict an argument of a term-level function to one type.

The `DataKinds` extension fixes this. With the extension enabled, the definition of `Nat` shown above now creates not only a type `Nat` with terms `Z` and `S n`, but also a kind `Nat` containing (empty) types `Z` and `S n`, the *promoted* versions of the equivalent data constructors. Where ambiguity arises, a constructor can be prefixed with `'` to indicate it is a promoted type constructor; for instance `[Int]` is the type of lists of integers as in standard Haskell, while `'[Int]` is the type-level list containing the single element `Int`. This way, we can turn arbitrary algebraic data types into user-defined algebraic kinds, which (unlike `Type`) are closed in the sense that they have only a certain, fixed set of constructors. The interaction between these closed kinds and type classes gives rise to the issue that this project aims to address, as shown in the example in the next section.

2.2.4 GADTs

Another recently added feature are Generalised Algebraic Data Types, or GADTs. While not strictly necessary to understand the problem at hand, this extension is frequently used for complex type-level programming, and we will use it for the example in (Section 2.3). When we define a data type using GADT syntax, we list the types of its constructors explicitly. For instance, rather than

```
data Maybe a = Nothing | Just a
```

we write

```
data Maybe a where
  Nothing :: Maybe a
  Just :: a -> Maybe a
```

Note that for a standard ADT, all constructors will have the same return type, and will have exactly those type parameters which also appear in the type definition itself (in this case the type has one type parameter, therefore so does each constructor). GADTs are types which do not have this property. For instance, they can have constructors whose return types are not polymorphic, even if the type itself is:

```
data F a where
  FInt :: F Int
  FAny :: a -> F a
```

Here, the constructor `F Int` can only be used for the case where `a` is `Int`. This means that when we pattern match on a value of type `F a`, the `FInt` branch can assume that `a` is `Int`:

```
f :: F a -> a
f FInt = 3
f (FAny x) = x
```

Thus, GADTs allow the programmer to link information about the structure of a value to its type.

The current implementation of GADTs was constructed via a significant modification to the `Core` language, creating a new theoretical language called System F_C [Sulzmann et al., 2007]. The key new feature are the new type-level objects *coercion* and *equality*. An equality `a` $\sim$ `b` represents the proposition that `a` and `b` are the same type. A member of `a` $\sim$ `b` is a coercion proof; it exists only if `a` and `b` are actually the same. These constructs allow GADTs to be translated into ordinary data types; for instance, the constructor `FInt :: F Int` becomes `FInt :: (a` $\sim$ `Int) => F a`.

2.2.5 Type families

Type families are functions from types to types. They were introduced in the form of *associated types* [Chakravarty et al., 2005], an extension of the functionality of type classes, before being generalised into a separate language feature in the form of *open type families* [Schrijvers et al., 2007] and further enhanced to support *closed type families* [Eisenberg et al., 2014], which are much more similar to term-level functions. While they behave similarly to type classes in some ways, our implementation concerns itself with type classes only; the possibility of extending our results to type families is discussed at the end of this report (Section 7.3.3).

2.3 Example

The following example showcases the use of these techniques.

Consider the function `head :: [a] -> a`, which extracts the first element of a list. This function throws an error at runtime if the list is empty, which is quite unsatisfactory as we would like to catch such errors at compile time. We could have a separate type for non-empty lists, but then we run into the same problem after removing one element. A more versatile solution would be to encode the length of a list in its type:

```
data Vec n a = ...
```

where `n` represents the length of the vector and `a` is the element type. We want n to be a natural number rather than an arbitrary type, so we will use the `DataKinds` extension in the manner shown above.

```
data Nat = Z | S Nat

data Vec (n :: Nat) a = ...
```

This now typechecks because `Nat` refers to a kind, rather than a type, so `n` is a type, rather than a term. So far we have left out the body of `Vec`. To give the `n` parameter its intended meaning, we want the type (of kind `Nat`) which is assigned to it to depend on the constructor used. For example, if we call the empty vector `VNil`, we want to have `VNil :: Vec Z a`, rather than `VNil :: Vec n a` for any `n`. Thus, we will make `Vec` a GADT

```
data Vec (n :: Nat) a where
  VNil :: Vec Z a
  (:>) :: a -> Vec n a -> Vec (S n) a
```

Now, `VNil` is the empty vector, with length 0, and the infix operator `(:>)` adds an element to a vector of length n, creating a new vector of length `S n`, that is to say, $n + 1$.

```
VNil :: Vec Z Int
1 :> 2 :> 3 :> VNil :: Vec (S (S (S Z))) Int
```

We can now define

```
vhead :: Vec (S n) a -> a
vhead (x :> _) = x
```

Since the type signature requires a vector of length `S n` for some `n`, writing `vhead VNil` will produce a compile time error, as desired.

We may want to compute the length of a vector at runtime. For a list, we would define:

```
length :: [a] -> Nat
length [] = Z
length (_ : xs) = S (length xs)
```

We could make the same definition for vectors, but this would mean that we have to "count" the elements each time we want to know the length, which is clearly inefficient. After all, the length of a vector is already known as part of its type, so we might expect to be able to write

```
vlength :: Vec n a -> Nat
vlength (_ :: Vec n a) = n
```

This code does not compile; the `n` in the type is a member of the kind `Nat`, whereas we want a term of type `Nat`. This is not an arbitrary restriction: An important feature of Haskell is type erasure, which means that all type level information is erased once typechecking is complete, and only term level data exists at runtime. Similarly to what we would do in a term-level setting, we can define a type class to circumvent this:

```
class IsNat (n :: Nat) where
  toNat :: Nat

instance IsNat Z where
  toNat = Z

instance IsNat n => IsNat (S n) where
  toNat = S (toNat @n)
```

The full type of `toNat` is `toNat :: forall (n :: Nat). (IsNat n) => Nat`; the expression `toNat @n` means that this `n` is passed as the implicit type argument `n` in the type of `toNat`, which is made possible by the extension `TypeApplications` and `ScopedTypeVariables`. This syntax is needed here because `toNat` has no arguments and does not use `n` in its return type, so there is no way for the compiler to infer the intended type[1]. A result of this is that uses of `toNat` look like a unary function application, except that the argument is a type rather than a term.

We can now write

```
vlength :: (IsNat n) => Vec n a -> Nat
vlength (_ :: Vec n a) = toNat @n
```

This type class is different in function from a typical type class whose argument is of kind `Type`. It is defined recursively with respect to the kind `Nat`; compare the instances above to the branches of a function like

```
f :: Nat -> ...
f Z = ...
f (S n) = ... f n ...
```

If we think of this type class as a function, we see that it is *total*: we cannot construct any `n :: Nat` for which `IsNat n` would not be satisfied[2]. Nevertheless, the type signature of `vlength` cannot be shortened; we have to write the constraint `IsNat n`, even though we know it will always be satisfied. In relation to the two mental models of type classes discussed earlier, we now have a discrepancy. To a programmer who thinks of `(IsNat n) =>` as a runtime argument of `vlength`, which it must be to preserve the data of `n` at runtime, this makes perfect sense. But if one thinks of `IsNat` as a predicate, as is typical when writing standard Haskell, the constraint in the type signature of `vlength` looks redundant. The comparison between this sort of type class and a regular function is also broken: we cannot simply call `toNat`, even though we know it is total, without first declaring our intention to use it in the type signature.

It is clear that the compiler's lack of distinction between these total type classes and others increases the mental overhead required when programming with them. This is a relatively minor issue in this simple example, but it is compounded greatly if the type class in question has a more complex argument type, or if multiple such type classes are used together. In programs that make heavy use of this style of programming, large parts of every function's type signature may consist of constraints that most programmers would intuitively think of as redundant.

Throughout the later chapters of this report, we will return to this example, considering a program where `vlength` is given the type signature `Vec n a -> Nat`, with no constraints, and our implementation allows the program to be compiled anyway. Note that type parameters are typically inferred by the compiler (for instance, we did not have to introduce the variables `n` and `a` in the signature above), but can also be specified explicitly using the `ExplicitForAll` extension. We could also write `vlength :: forall n a. Vec n a -> Nat`, with the same meaning. In the examples in this report, we often use explicit `forall`s for clarity.

2.4 GHC plugins

Our implementation is a GHC plugin which modifies the output of the GHC typechecker. To properly explain its functionality, it is necessary to first review GHC's compilation process and the relevant extension points for plugins.

[1] Allowing such a function to be defined requires the use of the `AllowAmbiguousTypes` extension

[2] There are exceptions to this; see Section 6.2.6

2.4.1 The compilation pipeline

Broadly, GHC compiles a Haskell program by executing the following steps [Marlow and Peyton Jones, 2012]:

1. The *parser* turns the source text into an Abstract Syntax Tree (AST).
2. The *renamer* finds out which occurrences of a variable name referto which binding and assigns each binding a unique identifier (of type `Unique`).
3. The *typechecker* converts type and class declarations to their `Core` equivalent, annotates variables with their types, and resolves type class constraints.
4. The *desugarer* converts the typechecked Haskell code into `Core` code (also typed).
5. Several optimisations are applied to the `Core` program
6. The optimised `Core` program passes through two more intermediate languages before being compiled into machine code; these are not important for this project.

This process is applied to each module individually; in particular, if module `A` imports module `B` then `B` must be compiled first to determine the types of any declarations that `A` imports.

2.4.2 Plugin extension points

GHC provides a plugin API that allows external code to hook into various steps of this process. Most importantly:

- *Constraint solver plugins* are executed during the typechecking process and can influence how type class constraints are solved; in particular, they can dynamically generate solutions to constraints for which no actual instance exists in the source code [GHC Team, 2023, Section 7.3.4]. These plugins run in the `TcPluginM` monad, which provides access to a subset of the compiler's API.
- *Typechecked source plugins* are executed directly after typechecking and can make arbitrary modifications to the typed AST before it is desugared [Pickering et al., 2019]. These plugins have access to the compiler's `TcM` monad, allowing them to invoke any of the typechecker's API functions.

Chapter 3

Implementation structure

In this chapter, we will discuss the overall structure of our implementation (Section 3.1) and explain its user-facing API (Section 3.2) as well as the constraint solver entry point (Section 3.3). The bulk of the implementation is described in the following two chapters.

3.1 Overview

The objective of the project is to create an extension of GHC which is capable of recognising type classes for which all possible applications match an instance head (*total* type classes), and to rewrite user code during compilation so that any function that requires a total type class constraint to be solved will compile even if the user did not write the constraint. Concretely, if `C` is a class taking a single parameter of (algebraic data)kind `T`, the plugin should first check that `C a` can be solved for every `a :: T`. If so, a function declaration like `f :: a -> ...`, where the body of `f` gives rise to a `C a` constraint, should compile, as though the user had written `f :: C a => a -> ...`. In the example from Section 2.3, the class `IsNat` should be recognised as total, and the type signature of `vlength` should be automatically changed to `forall n a. IsNat n => Vec n a -> Nat`.

Notably, these two steps are entirely separate in their implementation, connected only by the marker that a class is intended to be total. A user who only wants to use the checking behaviour can invoke it without triggering any modification to the compiled program, and a user who writes a class which they know is total but which does not pass the checker can override the check to perform the modification anyway. The second step could also easily be repurposed to be triggered by a different condition (see Section 7.3.2).

The first part is implemented by the *checker* (Chapter 4). The second part is implemented in two separate components. The *solver* (Section 3.3) is a small constraint solver plugin which can solve any constraint of the form `C ...`, where `C` is marked as total, by generating a "placeholder" evidence term which can be uniquely identified, but contains no actual information. It is also responsible for invoking the checker. The *rewriter* (Chapter 5) is a typechecker plugin which searches the typechecked code for generated placeholders and removes them, instead adding an appropriate constraint to the type of the function in question. This modified type is then propagated throughout the program, and the new constraints arising from the modified calls of the function are solved, possibly re-invoking the solver and generating more placeholders. The process then repeats until no more functions need to be modified. Finally, a small API is exposed to the user to control the behaviour of the plugin (Section 3.2).

3.2 User-facing API

The exposed module `TotalClassPlugin` provides definitions that can be used to control the plugin. The most important of these is the type class `TotalClass`, which accepts a single argument of the form `C :: a1 -> a2 -> ...Constraint`, i.e. a type class. The type of `TotalClass` is `forall ck. IsClassKind ck => ck -> Constraint`, where the `IsClassKind` constraint enforces that the argument is of the form above. This design allows the plugin to handle type classes with multiple parameters, introduced by GHC's `MultiParamTypeClasses` extension [GHC Team, 2023, Section

6.8.1]. `TotalClass` is an ordinary type class, which the user implements to mark that a class should be total. To do so, they must provide a term of type `TotalityEvidence C`, which acts as a "proof" of totality. This type has no exposed constructors, so such a term can only be generated using the functions provided in the module, of which there are two. The `checkTotality` function is a method of the `CheckTotality` type class, which has the same type as `TotalClass` but has no instances; calling this function will invoke the checker to perform the totality check. Alternatively, the `assertTotality` function can be used to override the totality check. This is needed if the programmer's class is too complex to be supported by the checker, or is itself a "magic" constraint whose instances are generated at compile time. In particular, GHC itself provides classes such as `KnownNat` (a more sophisticated implementation of the `IsNat` example from Section 2.3), which is functionally total, but has no instance declarations that the plugin can check. In cases such as this, an assertion must be used. The plugin itself exposes `TotalClass` instances for the relevant GHC classes. Finally, the `CheckTotalityResult` class performs the same check as `CheckTotality`, but makes details of the check result available through its methods rather than raising an error if the check is unsuccessful; this is primarily intended for testing.

3.3 The solver

The solver is a constraint solver plugin responsible for invoking the two main components. It contains a small amount of logic which classifies a wanted constraint depending on whether it has the form `CheckTotality ...` or `CheckTotalityResult ...`. For these, it invokes the checker; otherwise a corresponding `TotalClass` constraint is generated (unless the constraint being solved is itself a `TotalClass` constraint, in which case the plugin must fail immediately to avoid an infinite loop).

Essentially, the plugin functions like a "universal" instance `TotalClass c => c a1 a2 ...`, for any class `c` (note that this is not a valid Haskell instance declaration, because `c` is not a concrete class). However, instead of emitting `TotalClass c` as a new wanted constraint, the plugin re-invokes the solver to solve this constraint, and emits no solution at all if this fails. This preserves the normal "No instance for `c` ..." error message if there is no `TotalClass c` instance; otherwise, unrelated constraint solver failure would trigger the plugin and either produce a less helpful "No instance for `TotalClass c` ..." message or add a bogus `TotalClass c` constraint to an inferred type signature.

If the check succeeds, the plugin solves the original `c a1 ...` constraint by generating a "placeholder" `Core` expression to use as the evidence term. This takes advantage of the fact that the `Core` language, and thus the later stages of the compiler, do not distinguish between visible terms and type class evidence: if the placeholder instance is not removed during compilation due to a bug in the plugin, it will simply crash the program at runtime (just like e.g. a term-level `error` or `undefined` expression) rather than violating an invariant of GHC and crashing the compiler or producing undefined behaviour. The placeholder term contains a hard-coded string literal, which functions as both an error message to detect the aforementioned bugs and a way for the rewriter to identify such placeholders.

In the case of the `vlength :: Vec n a -> Nat` function from the example (Section 2.3), the use of `toNat` in the function body will give rise to an `IsNat n` constraint, which is not given in the type signature. Assuming the user has created an instance `TotalClass IsNat`, the plugin will solve this constraint, creating a placeholder term of type `IsNat n` which will be inserted into the typechecked AST.

Chapter 4

The checker

We present the totality checker, describing its general structure and invocation via the constraint solver (Section 4.1), the reporting of failed checks to the user (Section 4.2), and the three individual checks which must be performed: the exhaustiveness check (Section 4.3), the termination check (Section 4.4), and the context check (Section 4.5).

4.1 Invoking the checker

The checker is invoked whenever a constraint of the form `CheckTotality C` or `CheckTotalityResult C` needs to be solved. It performs three separate checks: the exhaustiveness check, which applies to the class `C` as a whole, and the termination and context checks, which are performed for each instance individually. These are called from the `check` function, which takes a `Bool` parameter specifying whether to fail when a check fails (`CheckTotality`) or return the three check results (`CheckTotalityResult`)

```
check :: Class -> Bool -> TcM (Bool, Bool, Bool)
```

The main solving function invokes this as follows:

```
res <- unsafeTcPluginTcM (setCtLocM (ctLoc ct) $ check cls (not get_result))
ev_term <- if get_result
  then mk_check_result_inst ck cls res
  else mk_check_inst ck cls
return $ Just (ev_term, ct)
```

Constraint solver plugins run in the restricted `TcPluginM` monad, but some of the checks require access to the full power of `TcM`, necessitating the use of `unsafeTcPluginM`. No issues were encountered with this during testing. If this were to be a problem, the checker could be easily reworked to use a placeholder-based approach similar to `TotalClass`, so that the constraint is always solved and the actual checking happens after typechecking when fewer restrictions are in place. In the `CheckTotalityResult` case (where `get_result` is `True`), the result of `check` is used to implement the instance methods providing access to the test results. Otherwise, an instance of `CheckTotality C` is created, which contains only the `TotalityEvidence C` carrying only type-level information. Note that in the latter case there is no need to inspect `res`, as `check` would have failed if the checks had not passed.

For a class `C :: k1 -> ...kn -> Constraint`:

- The exhaustiveness check verifies that each possible constraint `C T1 ...Tn` matches the head of an instance for `C`
- The termination check verifies that when the constraint solver successively applies the instances of `C`, it cannot enter an infinite loop
- The context check verifies that applying an instance will not give rise to any constraints for classes other than `C`

If all three of these properties are satisfied, then for any constraint `C T1 ...Tn`, the constraint solver will be able to apply an instance (by exhaustiveness), giving rise to new wanted constraints which are also of this form (by context), allowing more instances to be applied, a process which will terminate eventually (by termination). Thus, these checks are sufficient to ensure that the type class is total. The implementations of the checks presented here are sufficient to guarantee the properties; however, they are conservative in some cases, meaning that the checker may reject a class which is actually total.

4.2 Error reporting

GHC's error reporting system is implemented via the type class `Diagnostic`, which has instances for different components like the typechecker or the desugarer. The plugin emits errors via its own diagnostic type `TotalClassCheckerMessage`, which is then wrapped in a typechecker error to be emitted. Each possible cause of failure is equipped with a custom error message. Additionally, some of these messages are themselves wrappers around GHC messages, as discussed in the following sections.

4.3 The exhaustiveness check and the pattern match checker

The core idea of the exhaustiveness check is to take advantage of GHC's term-level pattern match checker, which solves a similar problem [Graf et al., 2020]. The details of this algorithm are irrelevant to this project; what matters is that the problem of checking that each possible value of a type is covered by a collection of patterns has already been solved effectively at the term level. In particular, the checker never produces false negatives; it may report an error where there is none, but never fails to reject a non-exhaustive match [Graf et al., 2020, Section 3.8]. Furthermore, total type classes are made possible by the `DataKinds` extension, which generates algebraic data kinds from types. As a result, each constructor of a data kind necessarily has a term-level equivalent, allowing for a straightforward translation[1]. Thus, given a class, the checker generates a function with parameter types corresponding to the parameter kinds of the class (`k1` to `kn` in the notation above). The function pattern-matches on its arguments, with the head of each instance of the class being translated into a pattern. If the function can be compiled successfully without the pattern match checker warning that the patterns are non-exhaustive, then the instance heads cover all possible input types, so the check succeeds.

4.3.1 Generating the evidence function

The function is generated using Template Haskell (TH), a GHC extension for metaprogramming [Sheard and Jones, 2002]. TH allows a Haskell program to generate Haskell source code, in a much simpler manner than by modifying the compiler, notably thanks to the *quasi-quote* notation which allows the user to write ordinary Haskell code which is not evaluated, but rather represented as an AST which the TH code can then operate on. Computations are written in the `Q` monad, which provides access to name generation among other functionality, and can be embedded into a program using the *splice* notation. This plugin, however, does not make a splice available to the user, but instead uses GHC's `runQuasi` function to lift code from `Q` into `TcM`, turning the TH code into GHC's representation for parsed source code, which can then be compiled.

The code generation is invoked via the function `mkEvidenceFun`, taking the name and arity of the class as arguments (the latter being slightly easier to extract in compiler code than in TH).

```
mkEvidenceFun :: Name -> Int -> Q [Dec]
mkEvidenceFun name arity = do
  let args = map (VarT . mkName . ("a" ++) . show) [1..arity]
  insts <- reifyInstances name args
  clauses <- mapM mkInstClause insts
  let fun_name = mkName ("evidenceFun" ++ nameBase name)
  return [FunD fun_name clauses]
```

[1] except in the case of the `TypeData` extension, see Section 6.2.5

This function looks up all instances of the class, turns them into definition clauses, and combines these into a function definition called `evidenceFunC`. The helper function `mkInstClause` decomposes the instance head (e.g. `C T1 ...Tn`), collecting the arguments and recursively turning them into patterns as follows:

- Type-level literals become their term-level equivalents
- Promoted data constructors become the corresponding data constructor (this is easy because they have the same name[2])
- Type-level lists and tuples are considered a special case of the previous point due to their unique syntax, and converted accordingly
- Variable patterns become wildcards

If any other elements are encountered in the type, an error is raised. In particular, this includes non-promoted type constructors, which cannot be translated to the term level. These correspond to matching on non-algebraic kinds like `Type` and `->`, which can never happen in a total class: there is no way to exhaustively pattern match on them because new members can be declared at any time, and there is no way a class can have both a specific match and a catch-all instance since type class instance heads cannot overlap in Haskell[3]. However, the checker will accept arguments of non-algebraic kind, provided they are not matched upon; consider for instance the following class:

```
class C (ts :: [Type])
instance C '[]
instance C ts => C (t : ts)
```

These instances perform structural induction over a type-level list of types, matching on the structure of the list but not inspecting the types within. This is clearly total and should be accepted. Indeed, the implementation of the conversion algorithm allows arbitrary nestings of kind constructors; as long as only algebraic kinds are matched on, the check can succeed.

The right-hand side of each clause is simply `()`; the function does not need to perform any computation as it will never be called. There is no need to generate a type signature for the function as the compiler will be able to infer it.

For the `IsNat` type class, the evidence function would be:

```
evidenceFunIsNat Z = ()
evidenceFunIsNat (S _) = ()
```

The inferred signature is `Nat -> ()`.

4.3.2 Compiling the function and extracting errors

The TH computation is invoked through the helper function `checkQuasiErrors`, which handles errors that arise from the TH code. Errors which are caused by `fail` calls in the code itself (this happens when the heads are not of the form required above) represent check failures, so they are returned to be shown to the user or suppressed in the case of `CheckTotalityResult`. Any other errors come from unforeseen problems with the code generation and cause the checker to fail entirely. The resulting TH declaration AST is turned into its GHC equivalent via the `convertToHsDecls` function from GHC:

```
ev_fun_binds <- case convertToHsDecls (Generated DoPmc) noSrcSpan ev_fun_dec of
  Left err -> failWithTc $ TcRnTHError (THSpliceFailed (RunSpliceFailure err))
  Right ev_fun_binds -> return ev_fun_binds
(group, Nothing) <- findSplice ev_fun_binds
(gbl_rn, rn_group) <- updGblEnv (\env -> env{tcg_binds=emptyBag}) $ rnTopSrcDecls group
((gbl_tc, _), _) <- captureTopConstraints $ setGblEnv gbl_rn $ tcTopSrcDecls rn_group
```

[2] [see `Note [Promoted data constructors]` in GHC Team, 2024, `compiler/GHC/Core/TyCon.hs`]

[3] GHC has an extension which bypasses this restriction [GHC Team, 2023, Section 6.8.8.4]. For simplicity, we do not support this for now; see Section 6.2.7

The `Generated DoPmc` parameter marks the code as being automatically generated, but signals that the pattern match checker should be run anyway. This may otherwise be undesirable for machine-generated code, but in the case of this plugin, running this is the main goal. The parsed code is split into a `HsGroup` (this mechanism handles TH splices in the compiled code, in this case there will be none) and renamed. The environment provided to the renamer (the call to `rnTopSrcDecls`) is that of the actual module being compiled, which ensures that the correct types are in scope (unless `checkTotality` is called in a module other than the one containing the instance declarations, in which case the user will have to ensure that the kinds and their constructors have been imported). However, the existing function declarations (`tcg_binds`) are removed as there is no need to compile them again. The resulting environment is then typechecked (`tcTopSrcDecls`), capturing and discarding any constraints emitted to avoid affecting the enclosing local environment (`caputeTopConstraints`). This process should not produce any errors; if there are any, the checker fails.

Finally, the typechecked bindings (`binds`) are desugared into `Core`; this is where the pattern match checker will be invoked.

```
checkDsResult cls $
  updTopFlags (flip wopt_set Opt_WarnIncompletePatterns) $
    initDsTc (dsTopLHsBinds binds)
```

Crucially, the warning for incomplete patterns is enabled for this computation, in case the user disabled it in the actual program. The `checkDsResult` function captures the warnings generated by the desugarer, turning the warning `DsNonExhaustivePatterns` into an error and failing the check while discarding everything else. If no such warning was emitted, the check succeeds.

4.4 The termination check and the Paterson conditions

The termination check is designed to reject type classes such as the following:

```
class NatNonTerm (n :: Nat)

instance NatNonTerm Z
instance NatNonTerm (S n) => NatNonTerm (S n)
```

Here, every type of kind `Nat` matches one of the instances, but instance resolution will never terminate in the `S` case. Unfortunately, detecting such cases in general is undecidable, since arbitrarily complex computations can be encoded as types.

By default, Haskell rejects instances that could lead to non-termination. The precise criteria are that each constraint in the context must satisfy the *Paterson conditions* [GHC Team, 2023, Section 6.8.8.3]:

1. Any type variable must occur in the instance head at least as often as in the constraint.
2. The total number of type constructors and variables (including repetitions) is lower in the constraint than in the head.
3. The constraint contains no type family applications.

The example above fails the second condition since the context and head are identical. The class `IsNat` complies with all three conditions; in the instance `IsNat n => IsNat (S n)` the head has one more constructor `S` than the constraint in the context. These conditions imply that each time the constraint solver uses an instance, the number of constructors and variables in a constraint decreases, ensuring eventual termination. This is by no means a necessary condition, and a user may wish to circumvent it. The `UndecidableInstances` extension allows this.

The plugin uses GHC's implementation of the Paterson conditions to check termination. The instances in question may have been compiled with `UndedicableInstances` turned on, so it is necessary to run the check within the plugin code. To do this, the GHC function `checkInstTermination` is called for each instance. Unfortunately, GHC does not expose this function, and the only function that calls it performs other checks that are not desired here, so we must duplicate the implementation in our code.

This check is conservative, in that a user may write a type class that they can tell is total, but that fails the Paterson check.

4.5 The context check

The third check is the simplest: the plugin decomposes each constraint in each instance's context, and if it is anything other than a class constraint for the same class being checked, the check fails. As per the definition, a class should be considered total if and only if its constraints can always be solved, which is not the case if some other constraint must be satisfied to apply its instances. Ideally, instances depending on other total type classes should be allowed, and the rewriter should insert these constraints as well where necessary. However, this functionality is not implemented in this check; a user could work around this by adding another constraint to the type of the class methods, rather than to the instance context. We revisit the possibility of loosening the restrictions of this check at the end of this report (see Section 7.3.1).

Chapter 5

The rewriter

In this section, we explore the design of the rewriter, which searches the AST for the placeholder instances inserted by the solver and modifies the program accordingly. We begin by explaining some relevant details of how GHC generates the typechecked AST (Section 5.1) and present the methods used to traverse and modify the AST (Section 5.2), then discuss the overall structure of the rewriting process (5.3) before finally diving into the details of the individual procedures (Sections 5.4 and 5.5).

5.1 The GHC typechecker and its output

5.1.1 Trees That Grow

GHC takes advantage of Haskell's (that is to say, its own) type system to safely reuse data type definitions between the parser, renamer and typechecker via a design pattern known as Trees That Grow (TTG) [Najd and Peyton Jones, 2017]. The main idea is that each data type is parametrised over an *extension descriptor* (we'll call it `p`) which indicates which variant is being used. Each type `T` has an additional constructor whose type is `XT p`, where `XT` is a type family. Likewise, each constructor `D` has an extra parameter of type `XD p`. By adding type family instances for `XT p` and `XD p`, additional fields or constructors can then be added to variation `p` only, while all other elements of the AST are shared. Thus, the types of compiler functions accurately reflect which of the extension-specific elements can appear in their arguments or return value. For instance, the function which typechecks a renamed expression takes a value of type `HsExpr GhcRn` and returns one of type `HsExpr GhcTc`, guaranteeing that any renamer-specific parts of the expression will be replaced with their typechecked variant (in particular, the untyped `Name`s become typed `Id`s).

5.1.2 The `TcM` monad and the typechecking environment

Each part of the compiler runs computations in a monad, providing access to IO among other things. The `TcM` monad used by the typechecker provides a global and local environment, containing the current state of the compilation. The global environment (`TcGblEnv`) contains persistent information, including the typechecked AST of definitions in the current module[1]. The local environment (`TcLclEnv`) tracks information specific to the current step of the compilation, such as which type variables are in scope in the expression being typechecked[2]. Information in the local environment is discarded when no longer relevant. Rather than modifying the environments within a computation, the `TcM` monad allows for a subcomputation to be invoked with modified environments, preventing accidental modifications.

5.1.3 The `HsWrapper` type

In the Haskell source syntax, functions often take invisible parameters, such as types and type class dictionaries, which are not written when the function is applied, nor are they explicitly mentioned in a pattern match. The typechecker is responsible for inserting these invisible elements, using

[1] [see the comments on `TcGblEnv` in GHC Team, 2024, `compiler/GHC/Tc/Types.hs`]
[2] [see the comments on `TcLclEnv` in GHC Team, 2024, `compiler/GHC/Tc/Types/LclEnv.hs`]

unification and constraint solving to find the right arguments to apply the function to. In the intermediate representation after the typechecker and before the desugarer, these are represented using the `HsWrapper` type. These indicate type-level information that modifies the type of the "wrapped" value[3]. For instance, the `WpTyLam` and `WpEvLam` constructors, appearing in polymorphic function declarations, represent abstracting over type and class variables respectively; they correpond to (possibly inferred) `forall a.` and `C a =>` binders, just like a term-level lambda `\x ->` corresponds to a `T ->` in the type. These constructors contain the name used to identify the introduced argument in other wrappers, including its `Unique`. Symmetrically, the `WpTyApp` and `WpEvApp` constructors appear when a function is applied to an invisible argument, and contain the concrete type, type variable, or evidence variable that the function is applied to. The constructor `WpLet` contains evidence bindings which assign specific type class instances or evidence variables to other variables. These can be either an immutable collection or a mutable variable to which bindings are added later on. The `WpHole` constructor contains no information (indicating no wrapper is needed), and the `WpCompose` constructor combines multiple constructors into one. There are other constructors, but these ones are of chief importance for us. Various helper functions are provided which simplify the structure of the produced wrapper; for instance, `mkWpLet` returns `WpHole` if the provided collection of bindings is immutable and empty, and the operator `<.>` composes its two arguments, but returns only one argument when the other is `WpHole`.

Wrappers occur in expressions in the form of the `WrapExpr` constructor, a `GhcTc`-exclusive extension constructor of `HsExpr`. This wraps a (possibly already partially applied) function, applying it to type and evidence arguments in the reverse order they appear in the function's type. Wrappers in function declarations are more complicated. They can appear in the extension field of the `FunBind` constructor, containing type and evidence abstractions as well as a `WpLet` binding type class evidence specific to that function. However, each mutually recursive group of bindings is additionally wrapped in an `AbsBinds`, which further modifies the types of the bindings. This records the result of let-generalisation, a mechanism which infers a maximally polymorphic type for functions with no type signature[4]. For each binding in the group, the `AbsBinds` contains an `ABExport` connecting it to its fully polymorphic type, which may include a wrapper. However, the `AbsBinds` also contains additional type parameters (`abs_tvs`), evidence arguments (`abs_ev_vars`), and evidence bindings (`abs_ev_binds`) which are shared by all functions in the group.

The function `vlength` will be compiled as a `FunBind`, wrapped in its own `AbsBinds` (since it is not mutually recursive) which adds no extra information (since it already has a user-written type signature and no generalisation is applicable).

5.1.4 Solving wanted constraints

Constraint solving in GHC proceeds as follows:

- When a polymorphic function is called, a wrapper is added to apply it to evidence variables (with new, random `Uniques`), and the constraints needed to fill in those variables are emitted (added to the local context)[5].

- When a new evidence variable enters the scope (for instance, when typechecking inside a function signature containing a `C a =>` binder), all of the constraints emitted while typechecking the contents of that scope are captured and collected, forming an *implication constraint* of the form $g \implies w$, where g is the new given evidence and w are the captured wanted constraints. A new mutable variable for evidence bindings is created and attached to the AST (`TcEvBinds`, often in a `WpLet`) and recorded within the implication. This implication is then emitted[6].

- At the top level (e.g. the top-level function declaration), the constraints emitted from the whole function are passed to the constraint solver, which attempts to check that the implications hold using declared type class instances as well as plugins[7]. The evidence terms generated by this, which describe how to compute each wanted constraint from its givens,

[3] [see the comments on `HsWrapper` in GHC Team, 2024, `compiler/GHC/Tc/Types/Evidence.hs`]
[4] [see `Note [AbsBinds] in` GHC Team, 2024, `compiler/GHC/Hs/Binds.hs`]
[5] [see `instantiateSigma` and `instCall` in GHC Team, 2024, `compiler/GHC/Tc/Utils/Instantiate.hs`]
[6] [see `tcSkolemiseGeneral` and `checkConstraints` in GHC Team, 2024, `compiler/GHC/Tc/Utils/Unify.hs`]
[7] [this happens at the call to `simplifyTop` in the body of `tcRnSrcDecls` in GHC Team, 2024, `compiler/GHC/Tc/Module.hs`]

are stored in the corresponding mutable variables. If the check fails, metadata attached to the implications is used to identify which part of the source code a constraint initially came from, generating the familiar "No constraint for ... arising from a use of ..." error message.

However, the constraint solver is also invoked when inferring the type of a function with no user-written signature[8]. Here, any constraints which cannot be solved are added to the inferred signature, rather than raising an error. The constraint solving process itself is identical in both cases; see Section 6.2.2.

5.1.5 Zonking

During typechecking, the AST can contain mutable variables, for example to contain evidence bindings which will be solved later, as discussed above. At the end of the typechecking process, a final pass is made over the entire AST to replace each mutable variable with its final contents[9]. In particular, a mutable `TcEvBinds` (constructor `TcEvBinds`) becomes an immutable collection of bindings (constructor `EvBinds`) This process is known as *zonking*. Notably, typechecker plugins are called after zonking and before desugaring, so the plugin can rely on being presented with a fully zonked environment.

5.2 Traversing the AST

All parts of the rewriter make extensive use of the library *Scrap Your Boilerplate* (SYB) to modify specific parts of the AST. The library provides a collection of combinators which extend a fully polymorphic function (usually `id` or `return`) with one that operates on a specific type [Lämmel and Peyton Jones, 2003]. Type classes (`Data` and `Typeable`) are used to retain the relevant type information at runtime. For instance, suppose we have a data structure `a :: A` which may contain one or more subterms of type `B`, and we wish to modify these using a monadic transformation `f :: B -> m B` for some monad `m`. Using SYB, we can achieve this by writing `everywhereM (mkM f) a`. Here, `mkM` extends `f` to a generic transformation which is equivalent to `return` for all terms that cannot be cast to type `B`, leaving them unchanged. The combinator `everywhereM` recursively applies `f` to the subterms of `B` in a bottom-up manner. It is a wrapper around `gmapM`, which applies a function to the immediate subterms of the input.

5.3 The loop

The rewriter performs two different types of passes over the typechecked AST. The first, called `rewriteBinds`, removes placeholder evidence terms and modifies the type of each modified function to include the needed constraints. This produces a collection of updated function `Uniques`, along with information about how their types changed. This data is passed to the `rewriteCalls` pass, which searches the AST for usage sites of the modified functions in the same module, updating their types, modifying the AST as necessary and solving the resulting constraints, which may cause new placeholders to be inserted. The rewriter then invokes `rewriteBinds` again, and the process repeats. The process terminates when `rewriteBinds` is completed without producing any updates.

The two functions are implemented in separate modules and each take a continuation which they invoke when done. The entry point of the renamer, which is provided as the `typeCheckResultAction` of the plugin, ties the two together:

```
totalTcResultAction :: [CommandLineOption] -> ModSummary -> TcGblEnv -> TcM TcGblEnv
totalTcResultAction _ _ gbl = do
  (gbl', _) <- rewriteBinds' (tcg_binds gbl)
  return gbl'
  where
    rewriteBinds' binds = rewriteBinds binds rewriteCalls'
    rewriteCalls' env binds = rewriteCalls env binds rewriteBinds'
```

[8] [this happens at the call to `simplifyInfer` in the body of `tcPolyInfer` in GHC Team, 2024, `compiler/GHC/Tc/Gen/Bind.hs`]

[9] [see `Note [Zonking to Type]` in GHC Team, 2024, `compiler/GHC/Tc/Zonk/Type.hs`]

The input to the process is the `tcg_binds` field of the global environment, containing the module's top-level function declarations (including type class methods). The process also updates the `tcg_type_env`, which is used by the compiler to quickly look up types. To ensure that all relevant information is extracted, the terminating case (in `rewriteCalls`) returns the final environments:

```
rewriteCalls ids binds cont
  | isEmptyDNameEnv ids = do
    printLnTcM "No new modified ids, ending loop"
    getEnvs
  | otherwise = ...
```

The local environment is returned for debugging purposes, although this is not used in the final code.

To establish that this algorithm terminates, suppose that every function `f` in the program has at least one of the following two properties:

1. `f` is not part of a mutually recursive declaration group.

2. If a constraint `C t1 t2 ...=>` is added to `f` by the rewriter, then any call of `f` instantiates `t1`, `t2` ... with either concrete types (e.g. `Z`) or type variables (e.g. `n`), as opposed to non-concrete applications like `S n`.

If the first property holds, then `f` will be modified by `rewriteBinds` at most once, since the functions which call `f` cannot be called by it; as a result, the calls of `f` will be modified at most once. If the second property holds, then for any call of `f`, either:

- The added constraints contain no free variables, in which case they should be solved via an actual instance rather than generating a new placeholder (assuming the class is actually total, of course).

- The added constraints are of the form `C a1 a2 ...`, where `a1`, `a2` ... are type parameters of the function; this generates a new placeholder if and only if this specific combination of type variables has not appeared in a generated constraint in a previous pass (otherwise that pass will have added it as a constraint parameter, which will be available as a given constraint in this pass).

Since each function has only finitely many type parameters (the plugin never modifies these) and there are only finitely many choices for `C` (since `TotalClass` instances are always written by the user), this guarantees that each call of `f` will be rewritten only finitely many times. Finally, the program itself contains finitely many function calls, so the assumption made above is a sufficient condition for termination. For a discussion of whether non-termination is possible in other cases, and whether such cases are likely to arise, see Section 6.2.1.

5.4 `rewriteBinds`

The `rewriteBinds` pass inspects the `AbsBinds` node which wraps each declaration group. It searches for placeholders in two places: the `abs_ev_binds` field, which is shared by the whole group, and the `fun_ext` wrapper of each individual `FunBind` in the group. If the former contains a placeholder, the `abs_ev_vars` field is augmented to contain the variable that the placeholder was assigned to, which has the effect of adding the constraint to the signature of every function in the group. The placeholder binding is then removed. Note that the intended type of the original constraint, including the arguments, can always be recovered from the placeholder due to the typed nature of the `Core` language, which requires each term to be annotated with its type. If a placeholder is found in a `WpLet` wrapper in the `fun_ext` wrapper, a new `WpEvLam` is added to that wrapper binding the variable in question, and the placeholder is removed. This adds the constraint to the type signature of that function. In our example `vlength :: forall n a. Vec n a -> Nat` from Section 2.3, the placeholder of type `IsNat n` generated by the solver will appear in the `WpLet` of `fun_ext`, so the plugin should change the type to `forall n a. IsNat n => Vec n a -> Nat`.

5.4.1 The insertion point

In either case, the new constraints are always inserted such that:

1. They are consecutive.
2. They appear after the `forall`s for all type variables that any of the new constraints refer to
3. The next part of the type signature after the last constraint is something other than a type variable (i.e. an existing constraint or an ordinary argument).
4. They appear as early as possible in the type signature while satisfying the other rules.

Condition 2 is clearly necessary, as the constraints would otherwise refer to an out-of-scope variable. Conditions 1 and 3 are chosen so that the correct insertion site is easy to identify in `rewriteCalls`; see the discussion in Section 5.5.3. Condition 4 ensures that a unique choice is possible. The conditions imply that the new constraints are inserted at the end of a sequence of consecutive `forall`s, at least one of which occurs in the constraints. In the `vlength` example, the last type variable appearing in the added constraint `IsNat n` is `n`, which appears in `forall n a.`, so the new constraint is inserted after this.

Implementing these rules for the `AbsBinds` case is easy: the new evidence variables are added to the start of the `abs_ev_vars`, so they will appear directly the last `forall` from `abs_tvs`. The `FunBind` case is more complicated because the user-written type signature could look like `forall a b. C a => forall c. T`. If the added constraint is `C b`, it must go between `b` and `C a` as per the rules. To find identify this location, the wrapper is traversed twice: On the first pass, the plugin inspects the right side of each `WpCompose` before the left, stopping when a type variable that occurs free in the new constraints is found.

```
findLastTyLamOfSet :: TyCoVarSet -> HsWrapper -> TcM TyVar
findLastTyLamOfSet vars w = case go w of
  Nothing -> ...
  Just tv -> return tv
  where
    go (WpCompose w1 w2) = case go w2 of
      Nothing -> go w1
      Just tv -> Just tv
    go (WpTyLam tv) = if tv `elemVarSet` vars then Just tv else Nothing
    go (WpFun _ w2 _) = go w2
    go _ = Nothing
```

The special case for `WpFun` is needed to support signatures like `() -> forall a. ...`, which are made possible by the `RankNTypes` extension.

The second pass goes from left to right, maintaining a state that reflects the progress of the traversal.

```
go :: HsWrapper -> Maybe TyVar -> Either (Maybe TyVar) (TyVar, HsWrapper)
go (WpCompose w1 w2) s = case go w1 s of
  Left s' -> case go w2 s' of
    Left s'' -> Left s''
    Right (tv, w2') -> Right (tv, w1 <.> w2')
  Right (tv, w1') -> Right (tv, w1' <.> w2)
go (WpTyLam tv) Nothing = if tv == target_tv then Left (Just tv) else Left Nothing
go (WpTyLam tv) (Just _) = Left (Just tv)
go (WpFun w1 w2 args) s = case go w2 s of
  Left (Just tv) -> Right (tv, WpFun w1 (w2 <.> rewrite WpHole) args)
  Left s' -> Left s'
  Right (tv, w2') -> Right (tv, WpFun w1 w2' args)
go _ Nothing = Left Nothing
go w' (Just tv) = Right (tv, rewrite w')
```

This helper function `go` implements this algorithm, receiving a state `Maybe TyVar` containing the current candidate for the final type variable and returning `Left Nothing` if no variable has been found yet, `Left tv` if `tv` is the last type variable encountered, and `Right (tv, w)` if the insertion has happened, with `w` being the updated input wrapper. Until the type variable `tv` found

in the right-to-left pass is encountered, no modifications are made, the input state is `Nothing` and the return value is `Left Nothing`. Once it is found, the function returns `Left (Just tv)` (the `go (WpTyLam tv) Nothing` case), which passes `Just tv` as the input state to the next iteration. The variable `tv` is replaced by each consecutive `WpTyLam` encountered from then on (the `go (WpTyLam tv) (Just _)` case). When the traversal reaches a component other than `WpTyLam`, the new `WpEvLam`s are inserted (the last case and the `WpFun` case, using a helper function `rewrite` whose implementation is not shown here) and the modified wrapper is returned. After this, no further state changes or rewrites occur (because the `WpCompose` case does not call `go w2` if `go w1 s` returned a `Right` result, indicating a rewrite happened), and the last variable seen (where the insertion occurred) is returned. If the application of `go` to the whole wrapper returns `Left tv` indicating that no more elements came after the `WpTyApp`s, the new `WpEvLam`s are inserted at the end (not shown here). Again, the `WpFun` clause handles the aforementioned special case.

5.4.2 Updating the type of a `FunBind`

In the `FunBind` case, the plugin must assign the function a new type signature computed from the updated wrapper. It is important to note that each type parameter of the function is introduced in two ways: the `forall` binder in the function's type signature, and the `WpTyLam` in the wrapper. While these are conceptually the same variable, the two do not have the same unique identifier. For instance, consider again the example `vlength :: forall n a. Vec n a -> Nat`. The `fun_ext` wrapper will be `WpTyLam m <.> WpTyLam b`, introducing new type variables `m` and `b`[10]. Occurrences in the function body (e.g. in a `WpTyApp`) will refer to `m` and `b` rather than `n` and `a`. The function body gives rise to a constraint for the total class `IsNat`, so the type of the placeholder variable will be `IsNat m`, and the wrapper will be `WpTyLam m <.> WpTyLam b <.> WpLet ((e :: IsNat m) = placeholder)`. This is the correct type for the `WpEvApp` that must be added to the wrapper, but if the same constraint was added to the type, the function would have the nonsensical type signature `forall n a. IsNat m => ...`, in which the variable `m` occurs outside of its scope. To fix this, the plugin uses unification, the same mechanism used by GHC to typecheck polymorphic functions, among other features. For types `A` and `B`, the unifier can compute a substitution which assigns the variables in `A` such that applying this substitution to `A` will produce `B`. The plugin proceeds as follows:

- The type of the (unmodified) function body is extracted. In our example, this will be `Vec m b -> Nat`.
- Using the `hsWrapperType` function provided by GHC, the resulting type when applying the unmodified wrapper to the body type is computed. Here this would be `forall m b. Vec m b -> Nat`.
- The list of type variables in this type is unified with the list of type variables in the function signature (we do not unify the whole types, which would already be considered equal since the differing variables are bound within them). In this case, the resulting substitution will be $m \mapsto n, b \mapsto a$.
- The placeholders are removed from the wrapper and replaced with a `WpEvLam` introducing the same variable, its type unmodified. In this case, the wrapper becomes `WpTyLam m <.> WpTyLam b <.> WpEvLam (e :: IsNat m)`.
- `hsWrapperType` is called again, now applying the new wrapper to the body type, which produces `forall m b. IsNat m => Vec m b -> Nat`.
- Finally, the computed substitution is applied to this type, yielding the final type signature `forall n a. IsNat n => Vec n a -> Nat`.

The only piece of information which is not preserved by the above construction of the new type is the `ForAllTyFlag` attached to each binder. This specifies how a type parameter interacts with explicit type application. To set these correctly, the plugin extracts the list of flags from the old type and sets the flag of each binder in the new type to the corresponding element of the list. Since the types match up exactly apart from the flags, this always works.

[10] In a real program, these would also be called `n` and `a`, but with a different unique from the variables in the signature. In examples like this one, we will use an entirely different name for clarity

5.4.3 ABExport

After the placeholders are removed and the new binders are added, the `ABExport` entry of each function in the group must be modified so that its fully polymorphic `Id` reflects the new type. For each entry, the type and evidence variables from the `AbsBinds` (the latter possibly modified due to placeholders in the `abs_ev_binds`) are added as `forall`s and constraints to the monomorphic type (possibly modified by the `fun_ext` wrapper update). Functions with non-trivial `abe_wrap` wrappers are currently not supported (see Section 6.2.4). This is the same logic used by GHC to compute the polymorphic types initially[11], ensuring that the resulting type will be the same as if the user had written the constraints to begin with. No let-generalisation applies to `vlength` since we provided a type signature, so the polymorphic type is the same as the monomorphic type in this case.

5.4.4 The generated UpdateData

Throughout this procedure, the updated types are stored in a mutable variable, which is made possible by the `TcM` monad. Its type is `DNameEnv UpdateInfo`, where `DNameEnv` is a type included in GHC, providing an efficient mapping from unique identifier names to other data. Each `UpdateInfo` is a record containing:

- The original type of the function.
- The new `Id` of the function, annotated with the rewritten type.
- The added constraints, mentioning the type variables bound in the new type; crucially in the same order in which they appear in the signature.
- The type variable after which the constraints were added in the signature.

Note that for each function binding, only the polymorphic variant produced by the `ABExport` is included in the update data; the monomorphic name is essentially local to the definition and does not appear in call sites[12]. In the case where a `FunBind` wrapper is rewritten, the substitution described previously is applied to the constraints and type variable in the `UpdateInfo` as well, as the variables occurring at call sites are those from the signature, not the wrapper. A call to `vlength` would have type `forall n a. Vec n a -> Nat`, not `forall m b. Vec m b -> Nat`, and must be updated accordingly. Thus, the update would have original type `forall n a. Vec n a -> Nat`, the new `Id` with type `forall n a. IsNat n => Vec n a -> Nat`, the constraint list `[IsNat n]` and the type variable `a`.

5.4.5 Sanity check

Finally, the plugin makes additional passes over the AST to identify any placeholders that were somehow missed. This is done in several passes: first to search for nodes that have a source location attached and also directly contain evidence bindings, such as `AbsBinds` and `FunBind`; this allows the plugin to indicate the relevant location in the error message. A final pass picks up placeholders that may occur without any location.

5.5 rewriteCalls

The `rewriteCalls` pass receives the `UpdateData` from the previous pass and updates the types of occurrences of the modified functions, terminating the rewriting process if there are none. Technically, the plugin would be functional without this feature provided that each rewritten function is defined in its own module. We deemed this option to be an unacceptable detriment to the developer experience.

We discuss the unique structure of the `rewriteCalls` traversal (Section 5.5.1), then discuss the management of the local environment (Section 5.5.2), the rewriting of call sites (Section 5.5.3), and the post-rewrite solving and cleanup of constraints (Section 5.5.4)

[11] [see `mkExport` in GHC Team, 2024, `compiler/GHC/Tc/Gen/Bind.hs`]
[12] [see `Note [AbsBinds]` in GHC Team, 2024, `compiler/GHC/Hs/Binds.hs`]

5.5.1 The recursive structure

The recursive traversal of the AST is much more complex in `rewriteCalls` because the places that signify that a modification needs to occur (variable expressions whose unique variables are mentioned in the `UpdateData`) are contained in a subtree inside the node where the modification will occur, instead of the other way around. Performing a bottom-up traversal is not enough, as the plugin could not easily differentiate rewrites which occurred in a sibling of the possibly modified node from those which occurred in its children. Instead, the plugin performs a top-down traversal called `rewriteCallsInBind`, stopping at each node that may need to be modified and starting a subcomputation for its subtree. This conditional recursion cannot be expressed efficiently using the combinators provided by SYB, so a new combinator is needed:

```
newtype M' m x = M' { unM' :: x -> m x }

extM' :: (Typeable a, Typeable b) => (a -> m a) -> (b -> m b) -> a -> m a
extM' def ext = unM' ((M' def) `ext0` (M' ext))

mkMMaybe :: (Monad m, Typeable a, Typeable b) => (b -> m (Maybe b)) -> a -> m (Maybe a)
mkMMaybe f = getCompose . extM' (\_ -> Compose (return Nothing)) (Compose . f)

orElseM :: Monad m => m (Maybe a) -> m a -> m a
orElseM f g = f >>= \case
  Just x' -> return x'
  Nothing -> g
```

The `extM'` function extends a generic transformation by one which may have a more specific type using SYB's `ext0` combinator, wrapping its arguments in a newtype solely to work around the lack of type-level anonymous functions in Haskell. It differs from SYB's `extM` only in that it does not require `m` to be a monad. The function `mkMMaybe` composes the provided monad `m` with `Maybe`, taking a type-specific function `f :: b -> m (Maybe b)` and extending it to return `Nothing` for other types. It calls `extM'` by combining `m` and `Maybe` into a single type constructor using Haskell's `Compose` utility type. Finally, `orElse f g` returns the result of `f` if it exists, or the result of `g` if `f` returns `Nothing`. Using `orElse` to combine `f` with an ordinary recursive traversal creates an operation which calls `f` wherever possible and recurses whenever `f` cannot be called or returns `Nothing`, but does not recurse into a subtree which `f` modifies succesfully, allowing `f` to control both modification and recursion.

The top-level traversal searches for three situations:

1. Function bindings, which can be top-level or nested in e.g. a `where` block, in order to update the local environment and add bindings from newly emitted constraints in the subtree; see Section 5.5.2.

2. Expressions in which type variables are introduced (e.g. by explicit type annotations) or functions are applied to type variables; the former case proceeds identically (1), while the latter is described in Section 5.5.3.

3. Variable expressions referring to functions which were modified according to the `UpdateData`, but which were not detected during (2).

The third case is an error, as the plugin cannot determine how to propagate the updated type in this case (see 6.2.4). If this happens, the error message will include the location of the variable in question; as a workaround, the user can explicitly cast the variable to its final rewritten type, a case which is handled appropriately by (2). This is implemented as follows:

```
rewriteCallsIn :: UpdateEnv -> GenericM TcM
rewriteCallsIn ids x = orElseM (mkMMaybe (rewriteLHsBind ids) x) $
                       orElseM (mkMMaybe (rewriteLWrapExpr ids) x) $
                       (mkM (noRewriteLVar ids) x >>= gmapM (rewriteCallsIn ids))
```

Whenever one of the three cases succeeds, no further recursion is needed as it is implemented within each case, but when all three fail, the traversal proceeds into the child nodes using `gmapM`.

5.5.2 Rewriting binders (again)

When the recursive traversal hits a function declaration, it must stop to perform several additional steps. In the case of `AbsBinds`, the type and evidence variables bound by the `abs_tvs` and `abs_ev_vars` must be taken into account when constraints are emitted in the inner bindings. This is implemented in the function `reskolemise`, which mirrors the second half of GHC's `tcSkolemiseGeneral`:

```
reskolemise :: [TyVar] -> [EvVar] -> TcM result -> TcM (TcEvBinds, result)
reskolemise [] [] thing_inside = do
  res <- thing_inside
  return (emptyTcEvBinds, res)
reskolemise tvs given thing_inside = do
  (new_ev_binds, result) <-
    checkConstraints (UnkSkol emptyCallStack) tvs given $
    tcExtendNameTyVarEnv (mkTyVarNamePairs tvs) $
    thing_inside
  return (new_ev_binds, result)
```

Here, the subcomputation `thing_inside` (which in this case will be the recursive traversal of the inner function bindings) is performed with a modified local environment which contains the new variables and wrapped in a call to the GHC function `checkConstraints`. This function captures any constraints emitted by `thing_inside` and constructs a new implication constraint whose givens come from the new evidence variables (corresponding to how GHC builds these constraints during typechecking, as we described in Section 5.1.4). This constraint is then emitted, to be captured by a previous layer or the top level. A new mutable variable is returned, into which evidence will be inserted when the constraint is eventually solved. In `AbsBinds`, this evidence is added to `abs_ev_binds`. This field actually contains a list of variables, but an inspection of the GHC source code reveals it never contains more than two entries. The reason for this is that in a type class instance, a group of bindings might need two variables to store evidence from different sources[13]. When the plugin runs, any variables already present must have been zonked previously (by GHC or by a previous `rewriteCalls` pass, see Section 5.5.4), so the plugin can retrieve their bindings and add them to the newly created variable, replacing the old immutable binding collection (or the second one, if there are two):

```
addToTcEvBinds :: TcEvBinds -> TcEvBinds -> TcM TcEvBinds
addToTcEvBinds (TcEvBinds _) _ = failTcM $ text "this should not happen"
addToTcEvBinds (EvBinds binds) new_ev_binds = case new_ev_binds of
  TcEvBinds (EvBindsVar{ebv_binds=binds_ref}) -> do
    updTcRef binds_ref (\ebm -> foldr (flip extendEvBinds) ebm binds)
    return new_ev_binds
  TcEvBinds (CoEvBindsVar{}) -> failTcM $ text "this should not happen"
  EvBinds new_binds -> return (EvBinds (new_binds `unionBags` binds))
```

The case for `FunBind` is the same, except that the variables are extracted from the `WpTyLam`s and `WpEvLam`s in the wrapper. The new binding variable is added as a `WpLet` to the end of the wrapper. If a `WpLet` is already present, the plugin proceeds as above, replacing its immutable contents with the new variable to which the existing bindings are added. The subcomputation passed to `reskolemise` is the traversal of the function body.

5.5.3 Rewriting call sites

In this section, we use a new example because `vlength` is too simple to showcase the edge cases our implementation covers.

When GHC typechecks a polymorphic function call, it splits off all the `forall`s on the outside of the function type, followed by all the constraints; these are turned into `WpTyApp` and `WpEvApp` wrappers, which are composed into a single wrapper (in reverse order, because the last wrapper component is the innermost) and added to a `WrapExpr`[14]. The inner expression is computed by typechecking the same renamed source expression, but with the type that remains after removing the

[13] [see `Note [Typechecking plan for instance declarations]` in GHC Team, 2024, `compiler/GHC/Tc/TyCl/Instance.hs`]

[14] [see `instantiateSigma` and `instCall` in GHC Team, 2024, `compiler/GHC/Tc/Utils/Instantiate.hs`]

aforementioned binders. This means that if we have `f :: forall a1 ...an. (C1 ..., ...Cm ...) => T1 -> T2` , a typical call `f x` compiles to `HsApp (XExpr (WrapExpr (HsWrap wrapper` f`))) `x, where `wrapper` is `(WpEvApp` e_m `<.> ...<.> WpEvApp` e_1 `<.> WpTyApp` t_m `<.> ...<.> WpTyApp` t_1`)` Here, f is the variable expression for the `Id` of `f`, x is the result of compiling `x`, e_k is the evidence variable to which the evidence for `C`k `...` will be assigned and the types t_k are determined by the type of `x` or the expected result type of the application. The type of `f x` is then `T2`, with each occurrence of `a`k replaced with the corresponding t_k. In this case, if the type of `f` has been updated to include additional constraints `(C' ..., ...C'm' ...) =>`, the plugin must introduce new evidence variables e'_k and replacing `wrapper` with `WpEvApp` e_m `<.> ...<.> WpEvApp` e_1 `<.> WpEvApp` $e'm'$ `<.> ...<.> WpEvApp` $e'1$ `<.> WpTyApp` t_m `<.> ...<.> WpTyApp` t_1. This is the reason for condition 3 in Section 5.4.1: because the new constraints are always at the end of a block of type parameters, their applications always belong directly before the first `WpTyApp` in a call wrapper, a location which can be found unambiguously by the rewriter.

For a concrete example, consider a function which takes two vectors of integers as parameters and trims the first to the length of the second, or return the second if the first is too short.

```
vtrim :: forall x y. (IsNat x, IsNat y) => Vec x Int -> Vex y Int -> Vec y Int
```

The implementation is irrelevant. Suppose that the two `IsNat` constraints were added by the plugin, and the function was applied to two vectors `v` and `u` of lengths 3 and 2 respectively. The typechecked function call will look like `HsApp (HsApp (XExpr (WrapExpr (HsWrap wrapper vtrim))) v) u`, where the wrapper is `WpTyApp 2 <.> WpTyApp 3`. The plugin will change the wrapper to `WpEvApp ey <.> WpEvApp ex <.> WpTyApp 2 <.> WpTyApp 3`. Crucially, the evidence variables need to have the specialised types `ex :: IsNat 3` and `ey :: IsNat 2`. However, the constraints recorded in the `UpdateInfo` for `vtrim` will have types `IsNat x` and `IsNat y`, where `x` and `y` are the unique variables from the declaration of `vtrim`, which are not in scope here.

Fortunately, the variable `vtrim` inside the function application always has the exact type (including `Uniques`) of the declaration, and `rewriteBinds` does not change the variables or their `Uniques`. Thus, to obtain the correct constraints, the plugin first computes the substitution which is induced by applying the original wrapper to the original variable, which can be done using the algorithm GHC already uses to implement the `hsWrapperType` function mentioned in Section 5.4.2[15]. For each binder in the original type which corresponds to a type or evidence application in the wrapper, this substitution maps the binder's variable to the argument it is applied to. In the example above, the substitution is $x \mapsto 3$ and $y \mapsto 2$. Applying this substitution to the constraints from the `UpdateInfo` will thus produce the correct types. Finally, the function `instCallConstraints` (also used by GHC itself for the same purpose) is called with these constraints, which creates the new evidence variables, emits the new constraints into the local environment (allowing them to be captured as described in the previous section) and returns the new wrapper.

Unfortunately, not all function calls are of the form described above, for the following reasons:

- If a type parameter appears after a constraint in the functions signature, the applications will be split into several wrappers, due to the recursive implementation in GHC explained at the start of this section. In the example above, if the user-written type signature was `vtrim1 :: forall x. IsNat x => forall y. Vec x Int -> Vec y Int -> Vec y Int`, then a call would have one `WrapExpr` with the applications to 3 and `IsNat 3`, wrapped in another with the application to 2; the plugin, when rewriting `vtrim1` during `rewriteBinds`, would be forced to insert `IsNat y` after `forall y.`, so the application to `IsNat 2` must be added to the *outer* wrapper.

- Using the `TypeApplications` extension, a function can be applied to a type argument explicitly, like `vtrim @3 @2 v u`. Such type applications are represented by `HsAppType` nodes rather than wrappers. If `vtrim` was called like this and the plugin had added both `IsNat` constraints to its type, a new `WrapExpr` with two `WpEvApps` would have to be wrapped around the outer `HsAppType`. If instead the signature was `vtrim2 :: forall x y. IsNat x => Vec x Int -> Vec y Int -> Vec y Int` and the plugin added only the second constraint, there would already be a wrapper containing the application to `IsNat 2`.

- Using the `RankNTypes` extension, the user can write a type signature like `vtrim3 :: forall x. IsNat x => Vec x Int -> forall y. IsNat y => Vec y Int -> Vec y Int`, where a

[15] Unfortunately, we had to copy this algorithm rather than calling GHC's API, as the GHC function discards the computed substitution and returns only the resulting type.

type parameter comes after a term-level parameter. Again, the plugin would have to add the new constraint after `forall y.`; the function application would consist of a wrapper, a function application, and another wrapper, and again the outer wrapper must be rewritten.

Due to these possibilities, the plugin cannot assume that the correct place to insert the new applications is in the immediate parent node of the variable itself. Instead, it is necessary to go through each nested wrapper, application or type application one by one; for function applications, the argument can be ignored. Wrappers encountered during this process can also introduce new type or evidence variables (if they come from a user-written type annotation); these are dealt with using the reskolemisation procedure described previously. If this process hits an inner expression which is neither one of the three types listed above, nor a variable, `rewriteCallsInBind` is called again, meaning that any modified function calls wrapped in another node (but not one of these three) will raise an error. See Section 6.2.4 for a discussion of whether this is always correct. If the innermost expression is a modified function, its `UpdateInfo` is passed back up the chain so that the evidence applications can be inserted.

On the way up the chain, the correct insertion point can be identified as follows:

1. To distinguish the correct wrapper from an unrelated one, the plugin computes the substitution induced by applying the wrapper to the unmodified inner expression, then checks if the substitution's domain contains the type variable recorded in the `UpdateInfo` of the function, which marks the correct insertion point.

2. The same trick works for the type application case[16]. Here, the plugin processes the previous wrapper and the type application inside it in one step.

3. Conditions 2, 3 and 4 from the insertion algorithm ensure that there cannot be an ordinary argument between the new constraints and the preceding type variable, so no rewriting needs to occur in the function application case.

In each case, the insertion is performed using `instCallConstraints`, as in the simple case. The substitutions from each type application (explicit or in a wrapper) are also combined on the way up, as the new constraints might contain type variables which are applied in different nodes and they must all be substituted. As a sanity check, an error is raised if any of the type variables in the added constraints are not in the domain of the computed substitution up to that point. It is crucial that the new constraints are emitted immediately rather than at the top of the chain, because the reskolemisation that should capture them may have occurred in the same chain. This happens in the type annotation case, where there will be at least three layers of wrappers: the inner one applies the function to variables introduced by the signature, the middle one introduces those same variables as per the annotation, and the outer one applies the annotated function to variables from the enclosing scope. For instance, if we write `(vlength :: forall m b. Vec m b -> Nat) (True :> VNil)` in the body of some other function, the wrappers will be `WpTyApp b <.> WpTyApp m`, `WpTyLam m <.> WpTyLam b`, and `WpTyApp Bool <.> WpTyApp (S n)` respectively.

5.5.4 Solving and re-zonking

The top-level `rewriteCalls` function captures all the constraints from the traversal and invokes the constraint solver:

```
(binds', lie) <- captureTopConstraints (rewriteCallsIn ids binds)
(gbl, lcl) <- getEnvs
new_ev_binds <- restoreEnvs (gbl, lcl) $ simplifyTop lie
when (any (isPlaceholder . eb_rhs) new_ev_binds) $ failTcM ...
```

The use of `restoreEnvs` ensures that the subcomputation interacts correctly with the enclosing context[17]. The evidence bindings returned from `simplifyTop` contain top-level evidence bindings, which do not contain any type variables from a local scope and can thus be reused by the whole program. It is helpful to check that the right-hand sides of these bindings (`eb_rhs`) contain no

[16] However, we call the helper corresponding to GHC's `piResultTys` directly rather than through `hsWrapperType`; a type application behaves identically to a wrapper with one `WpTyApp`.

[17] [see `Note [restoreLclEnv vs setLclEnv]` in GHC Team, 2024, `compiler/GHC/Tc/Utils/Monad.hs`]

placeholders, as that would imply that the class in question does not have an instance for some concrete constraint, so it is not actually total.

As previously discussed, the environment passed to the plugin is already zonked, meaning it contains no mutable variables. However, new mutable evidence binding variables must be created so that the evidence for constraints emitted from modified calls can be inserted correctly. Thus, at the end of the rewriting pass, these variables must be zonked again, replacing them with their final contents. Fortunately, it is known that all such variables occur in evidence bindings, so these can be targeted explicitly using SYB:

```
rezonkAllTcEvBinds x = orElseM (mkMMaybe (fmap Just . rezonkWpLet) x)  $
                       orElseM (mkMMaybe (fmap (Just . EvBinds) . rezonkTcEvBinds) x) $
                       gmapM rezonkAllTcEvBinds x
```

The special case for wrappers (first line) eliminates redundant `WpLet`s.

Finally, the updated environment is passed back to `rewriteBinds`, which will search for newly added placeholders.

Chapter 6

Evaluation

Our implementation reaches the goals we set out initially, accurately transforming a range of Haskell programs. To evaluate our achievement, we present a number of working examples (Section 6.1), then identify several limitations of our implementation and present possible solutions (Section 6.2).

6.1 Practical examples

6.1.1 Examples of correct check results

The checker is able to accurately accept type classes which feature recursive instances, multiple parameters, and combinations of algebraic and non-algebraic parameter kinds.

```
class IsNat (n :: Nat)
instance IsNat Z
instance IsNat n => IsNat (S n)
instance TotalClass IsNat where
  totalityEvidence = checkTotality

class TestMultiParam (x :: Nat) (y :: Nat)
instance TestMultiParam Z Z
instance TestMultiParam Z y => TestMultiParam Z (S y)
instance TestMultiParam x y => TestMultiParam (S x) y
instance TotalClass TestMultiParam where
  totalityEvidence = checkTotality

class TestNonADT (a :: Type) (n :: Nat) where
instance TestNonADT a Z
instance TestNonADT a n => TestNonADT a (S n)
instance TotalClass TestNonADT where
  totalityEvidence = checkTotality
```

The following is a more- sophisticated example, adapted from real-world code graciously provided to us by Nicolas Wu:

```
type Effect = (Type -> Type) -> (Type -> Type)

class Append (xs :: [Effect]) (ys :: [Effect])
instance Append '[] ys
instance Append xs ys => Append (x ': xs) ys
instance TotalClass Append where
  totalityEvidence = checkTotality
```

Here, the instances match on a type-level list whose members are not algebraic kinds.

When a class is rejected, an error message is produced detailing which condition was violated. This class does not have a `S Z` instance and produces the error "Exhaustiveness check failed: Pattern match(es) are non-exhaustive In an equation for 'the exhaustiveness check for `TestNonEx`': Patterns of type '`Nat`' not matched: `S Z`":

```
class TestNonEx (n :: Nat) where
instance TestNonEx Z where
instance TestNonEx n => TestNonEx (S (S n))
instance TotalClass TestNonEx where
  totalityEvidence = checkTotality
```

This class has a non-terminating instance and fails with "Termination check failed: The constraint '`TestNonTerm (S n)`' is no smaller than the instance head '`TestNonTerm (S n)`"':

```
class TestNonTerm (n :: Nat) where
instance TestNonTerm Z
instance TestNonTerm (S n) => TestNonTerm (S n)
instance TotalClass TestNonTerm where
  totalityEvidence = checkTotality
```

This class matches on an argument of kind `Type`, producing the error "Exhaustiveness check failed: Invalid type: `ArrowT`":

```
class TestNonADTBad (a :: Type) (n :: Nat) where
instance TestNonADTBad (Bool -> Int) Z
instance TestNonADTBad a n => TestNonADTBad a (S n)
instance TotalClass TestNonADTBad where
  totalityEvidence = checkTotality
```

Finally, this class has an instance depending on another constraint `Monoid a` failing with "Invalid constraint `Monoid a` in instance with head `TestCtxtBad a (S n)`":

```
class TestCtxtBad (a :: Type) (n :: Nat)
instance TestCtxtBad a Z
instance (TestCtxtBad a n, Monoid a) => TestCtxtBad a (S n)
instance TotalClass TestCtxtBad where
  totalityEvidence = checkTotality
```

6.1.2 Supported features for the rewriter

We tested the efficacy of the rewriter using a large collections of example functions using various combinations of Haskell features and requiring various modifications. The total class we use is GHC's `KnownSymbol` which turns type-level strings into term-level ones (via the method `symbolVal`), the string equivalent of our `IsNat`[1]. The test functions also make use of `Proxy` arguments, which allow a type parameter to be specified via a term-level argument, avoiding the use of `TypeApplications`. This allows us to test the common case of functions accepting arguments that fix the type parameters without introducing advanced features. In this section, we present an excerpt from this test suite showcasing some of the supported cases.

The following function calls the class method directly, so the constraint `KnownSymbol s` will be added to its signature:

```
testSimple :: forall (s :: Symbol). Proxy s -> String
testSimple x = symbolVal x
```

If this function is called by one which already has the constraint, the constraint solver will solve the wanted constraint emitted by the rewriter using the existing given constraint, so the following function's signature will not be changed:

```
testCall1 :: forall (s :: Symbol). KnownSymbol s => Proxy s -> String
testCall1 x = testSimple x
```

On the other hand, if the caller is also missing the constraint, it will be rewritten too:

```
testCall2 :: forall (s :: Symbol). Proxy s -> String
testCall2 x = testSimple x
```

The following function has two calls whose rewrites will emit the same constraint, so only one constraint is added to the signature:

[1]By using strings, we can instantiate each function with a type mentioning its name, producing clearer errors when debugging

```
testCalls2 :: forall (s :: Symbol). Proxy s -> String
testCalls2 x = testSimple x ++ " " ++ testSimple x
```

If the function's existing binders are in an unusual order, the new constraints can still be inserted:

```
testWeirdOrderCalls2 :: forall (s1 :: Symbol). KnownSymbol s1 =>
                        forall (s2 :: Symbol). Proxy s1 -> Proxy s2 -> String
testWeirdOrderCalls2 x y = testSimple x ++ " " ++ testSimple y

testArgBeforeTy :: () -> forall (s :: Symbol). Proxy s -> String
testArgBeforeTy () x = symbolVal x
```

Calls of these functions will still be rewritten correctly (via the `WrapExpr` and `HsApp` cases described in Section 5.5.3)

The called function can be specialised manually using a type signature or `TypeApplications`:

```
testMonoCastCall1 :: forall (s :: Symbol). KnownSymbol s => Proxy s -> String
testMonoCastCall1 (x :: Proxy s) = (testSimple :: KnownSymbol s => Proxy s -> String) x

testExpAppCall1 :: forall (s :: Symbol). KnownSymbol s => Proxy s -> String
testExpAppCall1 x = testSimple @s x
```

The call can also be in a nested function or lambda:

```
testNestedInferredCall1 :: forall (s :: Symbol). KnownSymbol s => Proxy s -> String
testNestedInferredCall1 x = goInferredCall1  x
  where
    goInferredCall1  x' = testSimple x'

testLambdaCall1 :: forall (s :: Symbol). KnownSymbol s => Proxy s -> String
testLambdaCall1 = \x -> testSimple x
```

The former also works if the inner binding has a user-provided type signature, which will also be rewritten if necessary.

Eta-reduced bindings (e.g. `f = testSimple`) at the top level or in a local declaration block are also supported.

The full test suite can be found in the file `test/TestModule.hs`. Using the interactive GHCi environment or an editor with language support for Haskell, the rewritten type of each binding can be viewed.

6.2 Known limitations

We present a number of deficiencies in our implementation and suggest improvements that could be made given more time.

6.2.1 Possible non-termination

In Section 5.3, we proved that the rewriter eventually terminates under a certain set of assumptions. These cases assumptions eliminate the possibility of a set of mutuallly recursive functions, where each rewrite introduces a new constraint that cannot be solved using those already inserted. We were not able to construct an example where this would happen, but could not rule it out. It is worth investigating if this is possible.

6.2.2 Inferred signatures

As described in Section 5.1.4, the constraint solver is invoked both when checking a function implementation against a type signature and when inferring a signature for a definition where none was provided. In the latter case, constraints which cannot be solved are added to the inferred signature rather than raising errors. When our plugin is active, it will also be invoked in this case, so that a constraint for a total class gives rise to a placeholder rather than appearing in the inferred signature. Of course, this placeholder will then be removed during rewriting, yielding the same

final signature that the function would have had if the plugin had done nothing. Unfortunately, the plugin is invoked in the same way in both cases. While some information about the context of a constraint can be deduced from the typechecking environment, we were unable to find a reliable method of determining whether a plugin invocation is part of checking or inference. If this is possible, it would enable a significant optimisation, since no extra rewriting pass would be needed to rewrite the occurrences of the unannotated function.

6.2.3 Scoped definitions

In the current implementation, each entry in the `UpdateData` is identified only by its unique name. In the `rewriteCalls` pass, the entire AST is searched for occurrences of these names. This is inefficient, as many of these names may belong to local definitions (e.g. in `where` bindings) which can only occur in a specific scope. Annotating entries with their parent scope could make the rewriting process more efficient, allowing the traversal to skip over subtrees where none of the names are in scope.

6.2.4 Unsupported cases in the rewriter

Since the plugin processes the typechecker output, it must make assumptions about what structure this will have. Not all possible arrangements of AST nodes are possible, and while individual cases are explained in the documentation or source code comments of GHC, there is no comprehensive breakdown of all possible structures available. To implement the current functionality, we examined the typechecked representations of many test programs and inspected the GHC source code to identify the corresponding generation code. Due to our limited time, there are a number of cases which we were unable to cover. Notably, all of our tests focus on ordinary functions; there is no special treatment of type class methods. Rewriting the signature of a method implementation is not possible since the class prescribes a particular type, but it would be helpful to have an explicit check and error message for this case to avoid unexpected behaviour.

There are a number of other cases which we never encountered in testing, but for which we were unable to investigate whether they can occur:

- `rewriteCalls` searches for places where a particular type parameter is bound, failing if an occurrence is found without one. This means that partially applied function calls could theoretically prevent the rewriter from triggering. We were unable to observe this issue in testing: even in an eta-reduced binding like `f = rewrittenFunction`, the typechecker inserts type abstractions and applications.

- `rewriteBinds` fails if a rewritten `ABExport` contains a wrapper, as we were unable to determine if such wrappers require special treatment. According to a comment in the GHC source code[2], these wrappers are only needed in rare cases.

- The `HsWrapper` type deals with functionality other than type and evidence parameters; notably, the constructors `WpCast` and `WpCoercion` are used for type coercions. We did not investigate these further, so it is possible that the algorithms dealing with wrappers (e.g. those in Sections 5.4.1 and 5.5.3) may fail if they are encountered. The `WpFun` constructor modifies the argument and result type of a function; here we support the special case of types like `() -> forall a. ...` (where `WpFun` is used to insert the type parameter after the unit argument), but not other cases.

Even if these cases are rare or impossible in practice, it would be worthwhile to investigate these further. Fortunately, the plugin's support for explicit type applications and type annotations should allow a user to work around any unsupported cases by either rewriting a signature themselves or force a call to appear with a predictable, full application. If this is frequently necessary, it may be possible to introduce a more convenient way to selectively disable the plugin, perhaps by providing a type class such as `NoTotal` with zero parameters, which is solved automatically and disables the solver when provided as a given constraint.

Due to the processing of the typechecker output, the plugin is also highly sensitive to changes to GHC's implementation, and would need to be carefully updated to support newer versions. The introduction of new type system features may also necessitate changes to the plugin.

[2] [see `Note [AbsBinds]` in GHC Team, 2024, `compiler/GHC/Hs/Binds.hs`]

6.2.5 TypeData

GHC's `TypeData` extension [GHC Team, 2023, Section 6.4.12] allows the user to define type-level algebraic data types directly, rather than promoting a regular type with `DataKinds`. While being semantically the same as data kinds, these types are unfortunately not supported by our checker, which relies on the unpromoted versions of the constructors being available for the exhaustiveness check. A possible fix might be to generate a term-level version of the kind during the check.

6.2.6 Irreducible type family applications

Partial type families can create types of any kind which do not resolve to one of that kinds declared member types. For instance, GHC provides the type family `Any`, which has kind `forall k. k`. Thus, a user could write a call of `vlength` where the input has type `Vec Any a`. This is unlikely to happen, since such type families will not be introduced unintentionally via type inference. However, this situation can cause the plugin to fail due to emitting a constraint with no free variables or add a constraint like `C Any a` to the function. To make debugging in this case easier, it would be better to try to detect this in the solver and produce a clear error message.

6.2.7 Overlapping instances

GHC allows the requirement that only one instance matches a particular constraint to be loosened [GHC Team, 2023, Section 6.8.8.5]. This allows the user to create a type class with a more general fall-back instance, such as:

```
class C (a :: Type)
instance C (Bool -> Int)
instance C a
```

The checker will reject this class since it matches on a non-algebraic kind, even though it is total. If this is an important use case, it could be supported by modifying the exhaustiveness check to accept such matches while not allowing them to contribute to exhaustiveness. For instance, the generated evidence function could look like this:

```
evidenceFunC x | isBoolToInt x = ()
evidenceFunC _ = ()
```

The guard function `isBoolToInt` cannot be implemented accurately since `Type` has no term-level equivalent, but the presence of a guard would cause the pattern match checker to mark this function as non-exhaustive if the second case were removed. The resulting error message would likely need to be customised further to communicate the reason for the failure to the user effectively.

6.2.8 Error reporting

The current implementation contains various sanity checks to detect rewriting errors that result from the aforementioned unsupported cases. As previously mentioned, these situations can often be worked around by the user. However, while the current error messages indicate the location of the issue where possible, the messages themselves are quite technical as they were initially intended only for debugging. GHC includes support for "suggested fixes" in diagnostics, which can also be provided by custom diagnostic types. Thus, a straightforward improvement would be to equip these errors with information about the suggested workarounds. For example, if the plugin encounters an unsupported wrapper structure in a declaration, it would be helpful to ask the user to rewrite the signature themselves.

Similarly, the code generated for the exhaustiveness check can fail to compile if the required constructors are not in scope as mentioned in 4.3.2. Since this can also happen during normal usage, identifying the specific errors that may arise and adding useful custom error messages would improve usability.

6.3 Alternative approaches

6.3.1 Rejected ideas for the overall approach

Early on, the following alternative implementation strategies were considered:

1. Using some other mechanism to make the correct instance available in a function at runtime, rather than modifying the source code to add another constraint.

2. Processing the renamed source code before it has been typechecked, in hopes of not needing to re-run parts of the typechecker.

3. Updating the source code immediately when a missing constraint is found rather than inserting placeholders to be searched for later.

The first option was ruled out quickly because the adding the constraints already provides the minimal amount of runtime information necessary to achieve the correct semantics. For instance, if the constraint `C a` is needed and the user does not write it, the resulting functions cannot actually have the semantics implied by its user-written type: since the choice of `a` is erased at runtime, there is no way to know which instance `C a` should be supplied at a call site. The only time this is not the case is if `a` is a constant rather than a type parameter, but then GHC would have already found the appropriate instance anyway. Thus, the choice of `C a` must be preserved at runtime, but that is exactly what adding the constraint does. Attempting to preserve this information some other way would likely duplicate existing functionality with little gain.

The second option is also not viable. It would certainly be possible to search the renamed program for occurrences of the methods of certain type classes, then for calls to the functions which contained them, and so on. However, as we saw in the current implementation, the way in which the resulting constraints are instantiated depends on the type of the function call. For example, if we see a call to `vlength` in the code the plugin can deduce that it may need to add `IsNat a` to the enclosing type signature, but to know the choice of `a` it would have to typecheck the expression anyway.

The third option seems attractive as it would perhaps eliminate the back-and-forth between all the components of the rewriter, but it is also infeasible. Due to the design of the GHC typechecker, which traverses the whole AST of a function first and solves all the constraints at the end, GHC never modifies the AST during constraint solving. Even if we could determine what change to make from within the constraint solver plugin (which may be difficult to begin with), it is thus likely that attempting to apply it would break invariants that other parts of GHC rely on. Furthermore, there would be no guarantee that all uses of the function would be typechecked after its declaration, so multiple passes would be needed regardless.

6.3.2 Rewriting the renamed program

After settling on the placeholder-based detection system, we also considered a design which adds the new constraint to the renamed source code and re-typechecks entire functions instead of modifying individual definitions and calls This is likely the most realistic alternative to the current implementation. The main benefit would be a much simpler and more reliable implementation; if the whole function is typechecked again, there is no need to search for individual nodes to modify, and the plugin need not make any risky assumptions about the structure of wrappers, nor handle each possible shape of a function application as a special case.

However, the performance of this approach may be significantly worse, since many unaffected parts of the program will be typechecked again, possibly many times since this approach would still require multiple passes. It may not even be possible to treat each function individually, as the typechecker is designed to process the whole program in one pass; trying to re-typecheck large chunks out of order may violate some other assumption. Furthermore, it would be necessary to traverse both the renamed and typechecked versions of the AST, finding the signature in the former which corresponds to a placeholder in the latter The issue of adding placeholders to inferred signatures becomes even more problematic here – if a constraint arose inside a function with an inferred type, the signature to be modified does not exist in the renamed AST. The plugin would have to turn the inferred signature into a renamed one and add it to the source code, necessitating another pass just like in the current implementation. At the same time, any bugs that arise from the conversion between the two ASTs may be difficult to diagnose and produce confusing error messages, since the failure would not be noticed until the next typechecking pass.

Therefore, to commit to this approach would have been to trade fine-grained complexity for a universal solution that either works without issue or fails completely, and we decided not to take this risk. In the long run, the current implementation is likely to be the more worthwhile approach, since it maximises performance by re-invoking the constraint solver when necessary and

never typechecking an expression more than once. Nevertheless, it would be insightful to implement both approaches and compare how large the gap in performance is.

6.3.3 The insertion point

The logic for where to insert the constraints into the function signature went through several iterations. A simpler approach would be to insert them at the very end of the invisible arguments (type parameters and constraints), but this would require special handling for edge cases like `() -> forall a. ...` (as the algorithm would then have to look past the `()` to search for more type parameters). To avoid having to cover all such cases explicitly, we decided to place the constraints as early as possible in the signature after the relevant type parameters. Initially we tried simply inserting them right after the last type variable appearing free in the constraints, but this makes call rewriting much more difficult: If the binders are `forall a b c.` and `C b` is to be inserted, all three type parameters will be bound in a single wrapper at the call site, and the plugin would have to identify the right one. Neither the wrapper itself nor the substitution it generates contain enough information to do this, since the `WpTyApp`s contain the type the function is applied to, not the unique of the parameter, and the former may not be unique (e.g. all of `a`, `b`, and `c` might be instantiated with `Z`). On the other hand, the current approach always selects an insertion point which is easy to find at the call site: the end of a block of type parameters, of which there is only one in each expression wrapper.

We also considered a very different approach to rewriting calls, which would be independent of the insertion point. Here, the plugin would compare the wrapper (or type application) and the old and new type of the inner expression. One binder/application at a time would be removed simultaneously from all three, until the outermost binders of the old and new types differ, which would mark the insertion point. This approach is arguably more robust, as it does not rely on how GHC constructs the wrappers. However, this would be much more complex to implement, and still require identifying the called function and passing that information up the tree to find out what constraints to insert. We thus decided that the benefits would not be worth the effort. If we were to discover more edge cases in the call expression structure that the current approach cannot handle, it may be worthwhile to revisit this idea.

Chapter 7

Conclusion

The use of type classes as total functions from types to terms is an unergonomic facet of type-level programming in Haskell, which can be alleviated using a plugin. Our implementation detects such total type classes using a conservative check which can be overridden when needed and rewrites code to allow constraints for these classes to be omitted, accurately rewriting a wide range of Haskell programs. Workarounds are available for programs that the plugin cannot process correctly.

7.1 Related work

7.1.1 Other obstacles to type-level programming

Bidirectional type class instances are a proposed solution to the issue of defining type class instances for certain GADTs [Pauwels et al., 2019]. For example, if a GADT has a constructor of type `FTuple :: F a -> F b -> F (a, b)`, a programmer may try to implement `Show F a` via the definition `show (FTuple x y) = show x ++ "," ++ show y`. However, this does not typecheck; GHC cannot deduce that `Show (a, b)` implies `Show a` and `Show b`, even though the only way to obtain the former instance is via the latter (due to the built-in instance with signature `instance (Show a, Show b) => Show (a, b)`). While this is another case where the compiler cannot deduce an instance that the programmer knows will exist, this issue appears to be orthogonal to this project. Crucially, the proposal's solution is to modify the elaboration of type class instances to include an inverse instance where appropriate, acting as though the instance above additionally defined instances `instance Show (a, b) => Show a` and `instance Show (a, b) => Show b`[1]. On the other hand, the key functionality of this project is to modify function types to request an instance even if the type annotation written by the programmer had no constraint at all. Thus, our goal could not have been achieved merely by automatically adding instances. However, if this proposal were to be implemented, it may complement our functionality to increase the usability of type classes in a wider range of type-level scenarios.

7.1.2 Constrained type families

Constrained type families are a proposed modification to the functionality of type families [Morris and Eisenberg, 2017], aiming to eliminate unintuitive, undesirable type families such as

```
type family Loop :: Type
type instance Loop = [Loop]
```

This type family has the unintuitive property that `Loop` is equivalent to `[Loop]`, which means that the compiler cannot generally reject an apparently nonsensical equality constraint such as `a ~ [a]`. For instance, the function `\x -> x : x` is, counterintuitively, well typed. This is one of many cases where unexpected complexity arises due to the power of type families. Morris and Eisenberg suggest that this can be fixed by requiring all type families to be associated with a type class. The constraint associated with the type class must then be satisfied to use the type family, preventing circuitous or otherwise nonsensical type families from being used. To recover the functionality of

[1] It should be noted that writing these instances explicitly does not work, because they would overlap with other instances

closed type families, which have all their definitions declared together, thereby allowing them to have overlapping patterns without ambiguity, the authors introduce closed type classes. While superficially similar to this project, they do not serve the same purpose; a total type class would also have to be closed, but closed partial type classes may exist.

The most relevant part of this paper is section 5.3, which explains how the proposed system would be improved if the compiler could identify a type class as total. While the focus is on type families rather than classes, there are similar ergonomics concerns as in this project. The authors propose that this feature should be implemented by allowing the compiler to consult an external totality checker. They suggest adding user-accessible syntax to mark that a type family should be total, or to override the result of the totality check, much like our implementation's API.

If this proposal were to be implemented, it would likely subsume our current approach to totality checking, but not the rewriting mechanism. Further investigation would be needed to determine how great the overlap would be, as the use of type classes as wrappers for type families may give rise to exactly those cases which our implementation cannot check accurately.

7.2 Singletons

The *singletons* library implements an alternative approach to simulating dependently-typed computation [Eisenberg and Weirich, 2012]. Instead of attaching a constraint like `IsNat` to a type parameter, functions accept a term-level argument such as `sn :: Sing n`, which encodes the choice of `n` at both the type and term levels. No constraint insertion is necessary in this case. However, it may be possible to integrate this functionality with ours. For instance, by using singletons in conjunction with the `constraints` library [Kmett, 2024], which allows constraints of type `c` to be wrapped in terms of type `Dict c` and manipulated explicitly at runtime, the evidence function generated by the checker could be given actual runtime functionality. In the case of `IsNat`, we would have `evidenceFunIsNat :: forall (n :: Nat). Sing n -> Dict (IsNat n)`, computing the correct instance from the input at runtime. The exhaustiveness check would then subsume the context check, since the generated function must use the existing instances to compute its output, and will be unable to do so if the instances require other constraints to be satisfied. This might make it easier to verify the correctness of the checker. However, it is unclear whether the runtime functionality is likely to be useful.

7.3 Future work

While the primary goals of the project were achieved, we have identified a number of promising directions for further development of the plugin, in addition to the known limitations we described already.

7.3.1 Additional checker features

The checker is the most obvious place for improvement, as many type classes that a programmer would think of as total are not identified as such by the plugin. A simple example are partially applied type classes. For instance, a type class like `C :: Type -> Nat -> Constraint` may have instances such that `C Bool n` exists for all `n`, but `C Int n` may not. In this case, a programmer may reasonably expect to be able to write `TotalClass (C Bool)`. Covering this case in the checker should be simple: the checker must consider only those instances of `C` matching `C Bool ...`. The rewriter should require no changes at all, as it gets all its information about the constraint types from the placeholders and never decomposes these types. The solver becomes slightly more complex: if the wanted constraint is `C T1 ...Tn a1 ...am` (i.e. at least the first n arguments are concrete), it no longer suffices to check if `TotalClass C` can be solved, we must also check for `TotalClass (C T1)`, `TotalClass (C T1 T2)`, and so on until `TotalClass (C T1 ...Tn)`.

This feature would enable a further extension in the form of conditionally total classes. Consider this class:

```
class C (a :: Type) (n :: Nat) where
  listOfMempty :: [a]

instance C a Z where
```

```
  listOfMempty = []

instance (Monoid a, C a n) => C a (S n) where
  listOfMempty = mempty : listOfMempty @a @n
```

The method computes a list of length `n`, using a `Monoid` instance on `a` to select a default element `mempty :: a`. Provided that `Monoid a` is satisfied, `C a n` exists for every `n`. Currently, this class fails the context check because the second instance has a constraint other than `C` in its context. It would be natural to be able to write `instance Monoid a => TotalClass (C a) where ...`. To support this, the solver would need to find any `TotalClass` instance that unifies with the required instance of `C`. Rather than requiring this `TotalClass` constraint to be solved completely (i.e. there are no unsolved residual constraints after calling the solver, see Section 3.3), the constraints from the instance declaration would be re-emitted, meaning that a placeholder is only generated when they are satisfied by the user-written type signature. Again, nothing would change in the rewriter. When the checker is invoked via `totalityEvidence = checkTotality`, the context of the `TotalClass` instance will become available as a given constraint (that is to say, the compiler will ask the plugin to solve `Monoid a => CheckTotality (C a)`), allowing the checker to use it and require that each instance's context can be obtained from this given. If the integration with `singletons` (see Section 7.2) were to be implemented, this would become easier: The type of the evidence function becomes `forall a (n :: Nat). Monoid a => Sing n -> Dict (C a n)`, and the function implementation uses each instance of `C`, causing the constraint solver to perform the necessary check as per its normal operation.

A further extension would be to allow other total type classes to appear in the context without being listed in the `TotalClass` instance, or even to generalise this by allowing the rewriter to add constraints to the context of any instance, not just the signature of a function.

A more ambitious goal would be to loosen the requirement that each instance conforms to the Paterson conditions. The general case is undecidable as previously discussed; it may be possible to use an external algorithm to attempt to verify termination, perhaps timing out if the problem is too complex (this is the approach suggested by Morris and Eisenberg [2017] for totality checking of type families).

7.3.2 Splitting up the plugin

The two core functionalities of detecting total type classes and rewriting programs to insert missing constraints automatically are mostly independent. If the latter feature were needed for some other purpose, our plugin could be refactored into two components, with the rewriter plugin being configurable to allow setting a different "marker". The checker plugin would export the `TotalClass` type class, and setting this as the marker would recover the current combined functionality.

7.3.3 Type families

Finally, the exhaustiveness check could be fairly easily extended to work for type families as well. However, the other checks would have to be redesigned since the form of a type instance definition is very different. If this extension were to be fully implemented, it should provide most of the checking functionality called for in the design for Constrained Type Families as well (see Section 7.1.2) [Morris and Eisenberg, 2017]

Bibliography

GHC Team. The Glasgow Haskell Compiler, 2024. URL https://www.haskell.org/ghc/download_ghc_9_8_2.html. Accessed June 2024.

Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1):11–49, April 2000. ISSN 1573-0557. doi: 10.1023/A:1010000313106. URL https://doi.org/10.1023/A:1010000313106.

Richard A. Eisenberg. Dependent Types in Haskell: Theory and Practice, 2017. URL https://arxiv.org/abs/1610.07978.

Sam Lindley and Conor McBride. Hasochism: the Pleasure and Pain of Dependently Typed Haskell Programming. *SIGPLAN Not.*, 48(12):81–92, September 2013. ISSN 0362-1340. doi: 10.1145/2578854.2503786. URL https://doi.org/10.1145/2578854.2503786.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937667. doi: 10.1145/1238844.1238856. URL https://doi.org/10.1145/1238844.1238856.

Simon Marlow and Simon Peyton Jones. *The Glasgow Haskell Compiler*. Lulu, the architecture of open source applications, volume 2 edition, January 2012. URL https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 60–76, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897912942. doi: 10.1145/75277.75283. URL https://doi.org/10.1145/75277.75283.

GHC Team. *GHC User's Guide*, 2023. URL https://downloads.haskell.org/~ghc/9.8.2/docs/users_guide/index.html. Accessed June 2024.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996. ISSN 0164-0925. doi: 10.1145/227699.227700. URL https://doi.org/10.1145/227699.227700.

Mark P. Jones. Type Classes with Functional Dependencies. In Gert Smolka, editor, *Programming Languages and Systems*, pages 230–244, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-46425-9.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 53–66, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311205. doi: 10.1145/2103786.2103795. URL https://doi.org/10.1145/2103786.2103795.

Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 53–66. ACM, January 2007. ISBN 1-59593-393-X. URL https://www.microsoft.com/en-us/research/publication/system-f-with-type-equality-coercions/.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, page 241–253, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930647. doi: 10.1145/1086365.1086397. URL https://doi.org/10.1145/1086365.1086397.

Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel Chakravarty. Towards open type functions for Haskell, January 2007. URL https://www.microsoft.com/en-us/research/publication/towards-open-type-functions-haskell/. Presented at the Implementing Functional Languages workshop, Sept 2007 (IFL07), but not part of its post-refereed proceedings.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. volume 49, page 671–683, New York, NY, USA, January 2014. Association for Computing Machinery. doi: 10.1145/2578855.2535856. URL https://doi.org/10.1145/2578855.2535856.

Matthew Pickering, Nicolas Wu, and Boldizsár Németh. Working with source plugins. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 85–97, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi: 10.1145/3331545.3342599. URL https://doi.org/10.1145/3331545.3342599.

Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. Lower your guards: a compositional pattern-match coverage checker. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi: 10.1145/3408989. URL https://doi.org/10.1145/3408989.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002. ISSN 0362-1340. doi: 10.1145/636517.636528. URL https://doi.org/10.1145/636517.636528.

Shayan Najd and Simon Peyton Jones. Trees that grow. *Journal of Universal Computer Science (JUCS)*, 23:47–62, January 2017. URL https://www.microsoft.com/en-us/research/publication/trees-that-grow/.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, page 26–37, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136498. doi: 10.1145/604174.604179. URL https://doi.org/10.1145/604174.604179.

Koen Pauwels, Georgios Karachalias, Michiel Derhaeg, and Tom Schrijvers. Bidirectional type class instances. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 30–43, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368131. doi: 10.1145/3331545.3342596. URL https://doi.org/10.1145/3331545.3342596.

J. Garrett Morris and Richard A. Eisenberg. Constrained type families. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi: 10.1145/3110286. URL https://doi.org/10.1145/3110286.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, page 117–130, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315746. doi: 10.1145/2364506.2364522. URL https://doi.org/10.1145/2364506.2364522.

Edward A. Kmett. constraints: Constraint manipulation, 2024. URL https://hackage.haskell.org/package/constraints-0.14.2. Accessed June 2024.