Type-Level Computation & The Road to Dependent Haskell

Masters' Thesis

Hazim Sager

Department of Computer Science
Imperial College London

2025

Abstract

This thesis explores two contributions toward a more expressive, pragmatic vision of Dependent Haskell, centred on type-level computation.

First, this thesis begins by addressing a long-standing limitation of the Haskell type system: the rigidity of type-level programming. To this end, a discussion is presented on the theory of unsaturated type families, with the intention of allowing partial application at the type level and bringing type-level computation closer in spirit to term-level programming. Then, an exploration into two implementation strategies is presented, of which one reveals latent sensitivities in the Glasgow Haskell Compiler (GHC) runtime while the other proves robust in backwards compatibility.

Later, this thesis presents System FO, a polymorphic typed lambda calculus that achieves what has long been considered contradictory: an explicit fix-point combinator within a decidable type system. This is accomplished through type-order stratification and, in similar spirit, the restriction that any countably infinite set of base types must admit a well-quasi order.

The fix-point combinator enables natural encoding of non-primitive recursion patterns at the type-level, making System FO an ideal compilation target for dependently-typed languages whose recursion patterns resist traditional termination checking techniques. This thesis also presents a variety of other avenues for future work opened up by the introduction of System FO, including automated proof search and aggressive compiler optimisations.

Collectively, these contributions demonstrate that the flexibility of term-level computation can be meaningfully and pragmatically mirrored at the type level, marking significant progress toward a fully Dependent Haskell.

Acknowledgements

I would like to express my deep gratitude to Dr Jamie Willis and Dr Nicolas Wu for supervising this project and for lending their expertise. Also, I am grateful for Grazia Troiano and Caroline Gilchrist for their support throughout my university career. I would also like to thank David Davies for proofreading my formalisation drafts.

I am especially grateful to my friends for their support and encouragement, especially to those who patiently listened to my many ramblings and discussions about this project.

Above all, I thank God for the strength and clarity when I needed it most.

Contents

Chapter 1

Introduction

"The difficulty in philosophy is to say no more than we know."
—Wittgenstein, The Blue Book

Haskell's type system is famously expressive. Just as famously, it is incomplete (Peyton Jones et al. 2007; Hinze 2003; Marlow et al. 2010; Eisenberg 2016). It permits a large amount of type-level computation but only through a patchwork of new extensions, experimental features and informal conventions ((Schrijvers et al. 2008; The GHC Team 2025; Yorgey et al. 2012)).

In some ideal world, we might hope that all the programs we write are correct. The type system we inherit from GHC is, on the whole, rather good at enabling this. At our disposal are a surprising range of *type-level proofs* of properties we might care about in our program (S. L. P. Jones, Washburn, and Weirich 2004; Jhala 2014). To start, consider a simple example:

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

While basic, we now know that add returns only an integer. But, perhaps unsurprisingly, types can capture much more than just the return type of a function. Our goal is to prove more complex invariants about the programs we write. Consider, for example, the following implementation of append.

```haskell
append :: List n a -> List m a -> List (Add n m) a
append (Cons x xs) ys = Cons x (append xs ys)
append Empty ys       = ys
```

The type alone guarantees that if the first list has length n and the second list has length m, then their concatenation through append has length $n + m$. By carrying the length as part of the type of the list, we allow the type checker to validate our work.

Yet, there is a gap between the expressivity of terms and the expressivity of types. At the term-level, partially applied functions compose naturally:

```
append :: [a] -> [a] -> [a]
append [] ys     = ys
append (x:xs) ys = x : append xs ys

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

appendToList :: [a] -> [a]
appendToList = map (append [1,2])
```

This compositionality is rather important for expressive programming; a program is often best specified as the composition of small and simple functions. At the type-level, however, this expressivity breaks down.

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

type family Map (f :: a -> b) (xs :: [a]) :: [b] where
    Map f '[]       = []
    Map f (x ': xs) = f x ': Map f xs

type AppendToList someList = Map (Append '[1,2]) someList  -- Illegal
```

This limitation dramatically hampers the expressivity of type-level computation. Due to this limitation, the use of *defunctionalisation* as a workaround is necessary, rendering code unergonomic (Reynolds 1972). To this end, this thesis describes both the theoretical elements necessary to support unsaturated type families in Haskell, presenting two theoretically sound approaches. Furthermore, the practical considerations for an implementation are presented for each approach, highlighting latent sensitivities of the Glasgow Haskell Compiler (GHC) architecture before culminating in a robust, full prototype implementation.

Where type-level computation becomes more complex, it would be ideal to guarantee that the compiler will be able to always type check a program. In context of a fully Dependent Haskell, the risk that this checking process is undecidable looms; simply allowing type-level computation to mirror the term level in power would immediately render Dependent Haskell undecidable and the type system unsound (Eisenberg 2016; Weirich, Voizard, et al. 2017).

This thesis therefore also explores the boundary of decidability of β-equivalence of typed lambda calculi. Through System FO, a novel type system that admits a fix-point combinator, this thesis achieves something seemingly contradictory: decidability despite fix-point combinators. System FO achieves this through two principled restrictions. The first is simple: types that do not admit a well-quasi order are eliminated. The second has little practical impact: higher-order behaviour is restricted such that a function may not take an argument of higher order than itself, with defunctionalisation acting as an apt workaround when needed (Reynolds 1972).

Together, the language remains expressive, decidable and supports both static and online verification of termination.

The implications extend beyond pure theory. System FO provides a compilation target for dependently-typed languages whose recursion patterns resist size-change termination analysis, or do not fit as a recursion scheme. Likewise, the language enables automated proof search in previously undecidable domains, and suggests new avenues for compiler optimisations that can aggressively eliminate dead code, partially evaluate programs and unroll loops. The language remains expressive enough to encode a rather large set of non-primitive recursive programs and yet remains decidable.

This thesis presents a set of theoretical contributions and practical considerations for GHC, aimed at enabling this style of programming:

- Chapter 2 covers the relevant background and research toward a dependently-typed Haskell.
- Chapter 3 introduces the theoretical contributions towards unsaturated type families and their place in the future evolution of the language.
- Chapter 4 addresses the practical design and engineering concerns involved in realising these ideas within GHC.
- Chapter 5 offers a formalisation of System FO.
- Finally, Chapter 6 offers an evaluation of the work and explores avenues for future research.

Chapter 2

Background

"We are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances."
—Issac Newton, Principia Mathematica

In this chapter, we survey the landscape of type-level computation in Haskell: its strengths, its limitations, and the particular tensions that arise when types are asked to behave like terms. Our goal is to isolate and make explicit what the type system we inherit from GHC leaves wanting and determine what the solutions are.

2.1 The Shape of Type-Level Computation

Over time, a series of extensions and new features in GHC have expanded the expressiveness of type-level computation, allowing programmers to encode increasingly complex invariants in their programs (Peyton Jones et al. 2007; Hinze 2003; Marlow et al. 2010; Eisenberg 2016; Schrijvers et al. 2008; The GHC Team 2025; Yorgey et al. 2012). As the line between terms and types blurs, we arrive at a natural consequence: we must learn how to compute at the type-level just as we do at the term level.

We begin, then, by pinning down what it means to compute at the type-level in Haskell. Types, like `Int` or `Char`, are collections of values and act as stand-alone symbols in type-level computation. While they describe collections of terms, and are interesting because they constrain the set of terms that a program might hold in a given place in memory, it will be beneficial to consider these the basic building blocks of type-level computation.

Types belong to a collection called `Type` (sometimes written as `*`). The collection `Type` is referred to as a kind. As types classify terms, and constrain the set of terms, so too do kinds classify types. Naturally, these relationships lead us to begin to ask what structures classify kinds and what structures might classify those and so on, and an answer is found in traditional dependent type theory: the hierarchy extends indefinitely upwards (Martin-Löf 1980).

But in Haskell, this hierarchy is collapsed into a single *impredicative* universe governed by the axiom `Type : Type` (Weirich, Hsu, and Eisenberg 2013). This choice simplifies the language, but at a cost: our type system now is logically inconsistent. In a *fully stratified* type theory, types are layered into distinct universes to avoid paradoxes . Haskell, by contrast, accepts this axiom as a pragmatic compromise. For our purposes, this is a crucial point. Haskell is not, by no uncertain terms, a proof assistant. We work within a system designed to support programming first, and logical consistency second.

Having outlined the basic structure of types and kinds in Haskell, this thesis turns to the mechanisms that enable computation at the type-level. Broadly, type-level functions arise through three main constructs: *data types*, *type families* and *generalised algebraic data types* (GADTs). The simplest of these is the data type.

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

A data type allows us to produce a new type from a given type in a fixed and structural way. In the case of `Maybe a`, we begin with a type `a` and produce a new type with values that are either `Nothing` or the values of `a`, constructed by the data constructor `Just`. As kinds classify types, and `Maybe` is a type-level function, we expect `Maybe` to have the *kind* signature `Maybe :: Type → Type`. Similarly, a type-level function with two arguments might have the kind `Type → Type → Type`.

```
-- Either :: Type -> Type -> Type
data Either a b = Left a | Right b
```

Data types provide algebraic structure, with some important consequences later on, but do not compute with much flexibility. To introduce general computation to the type-level, the language provides type families: type-level functions that arbitrarily map input types to output types. Type families are, notably for this thesis, a type-level representation of a function and are hardly distinct to term-level functions in their role in computation.

```
not :: Bool -> Bool
not True  = False
not False = True

type family Not (a :: 'Bool) :: 'Bool where
    Not 'True  = 'False
    Not 'False = 'True
```

Here, it is quite easy to see the parallel between term-level functions and type families. Both `not` and `Not` express, for all intents and purposes, similar semantics. At the type-level, the use of the quote (`'`) operator before a type or a data constructor refers to the *promoted* form of that structure. Promotion lifts types to kinds, and data constructors to types, allowing term-level structures to be reinterpreted at the type level.

Type families are pure calculators at the type-level; they compute types but do not guard the structure of terms. Generalised algebraic data types fill this gap by tightly coupling invariants into the structure of a type.

```
data Vec (n :: Nat) a where
    VEmpty :: Vec Zero a
    VCons  :: a -> Vec n a -> Vec (Successor n) a
```

In the example of `Vec`, the output type of the data constructor `VCons` depends on the nature of the second input type. This is, in essence, *type refinement* based on the input types and allows us to encode invariants based on the precise refinement chosen. The type checker has been informed that after applying `VCons`, the resulting vector has a length one larger than the input vector. This is especially useful. The type checker is now able to make sure that our programs that use `Vec` are, in some ways, sensible. Take the following example:

```
headVec :: Vec (Successor n) a -> a
headVec (VCons x _) = x
```

The objective of `headVec` is to return the first item in a `Vec`. Occasionally, our `Vec` might be empty and if not carefully implemented (as is the case for the standard library `head`), the function may become partial in this edge case. Here, in using the signature `headVec :: Vec (Successor n) a → a`, the type checker makes sure that the programmer only ever uses `headVec` in a `Vec` that contains at least one item.

Data types, type families and GADTs together form the mechanism for type-level computation in Haskell. They allow types to carry structure, to compute and to express invariants. As we push the boundaries of what types can express, cracks begin to show: not everything can be promoted, higher-order functions are conspicuously missing, and invariants often must exist tightly coupled into the type-level definition of a data structure. In the sections that follow, this thesis turns to these tensions directly.

2.2 Unsaturated Types & Matchability

Type families behave almost like term-level functions. They take arguments, they produce results and they underlie the mechanism of computation. But unlike term-level functions, they reject partial application.

When computation becomes complex, we often find our programs manipulating functions as first-class entities. Yet the type system GHC provides offers little ability to manipulate *unsaturated* - that is, partially applied - type families. An ad-hoc mechanism, offered today, to work with a representation of unsaturated type families is the unwieldy process of *defunctionalisation* (Reynolds 1972).

Defunctionalisation is the affair of assigning opaque symbols to partially applied functions. Just as `Either` is the symbol representing the computation underlying `Either a b`, we may choose other symbols to represent `Either Int`, `Either Char` and so on. And, just as we give the fully saturated form of `Either` semantics through its corresponding definition, we must find a way to give semantics to the symbols we have chosen for the unsaturated forms.

To assign semantics to those symbols, we rely on a type-level interpreter. The *Singletons* library provides an example of such an interpreter, as in the following example:

```haskell
type (:->:) = a -> b -> *
infixr 0 (:->:)

type family Id (n :: a) :: a where
    Id n = n

data IdSym = * :->: *

type family Apply (a :->: b) (x :: a) :: b where
    Apply IdSym n = Id n
```

We consider `IdSym` the symbol representing the unsaturated form of `Id`, applied to no arguments. Once defined, we can manipulate `IdSym` as if it were a first-class entity. Later, when we want to apply it onto an argument, we use the interpreter `Apply`. This is a rather circuitous way of treating `Id` as a first-class entity, but in the absence of native support for unsaturated types, we are starved for choice.

At present, Haskell sometimes assumes that type-level functions are more well-behaved than they later betray themselves to be: that they are often injective enough, generative enough and, crucially, free enough of higher-order behaviour (Eisenberg 2016; Kiss 2018; Chen 2024). This assumption is largely safe today because type families cannot be partially applied. But it leaves the type system precariously exposed the moment we allow functions to be unsaturated:

```haskell
type family BitWidth (x :: Type) :: Nat where
    BitWidth Int8 = '8
    BitWidth Char = '8

putState :: (f a ~ f b) => a -> State b ()
putState x = put x

happy :: ([a] ~ [b]) => a -> State b ()
happy x = putState x
```

The `BitWidth` type family is an example of a *non-injective* type family - `Int8` and `Char` are mapped to the same output. Meanwhile, `putState` is an example of a function that relies on a constraint where `f` is quantified unsaturated. In today's Haskell, these definitions compile without incident. The type checker assumes that a type family cannot actually ever appear unsaturated and so it deems that if we satisfy $f\ a \sim f\ b$, the constraint is decomposable into $a \sim b$. But the moment we allow unsaturated type families, this assumption quickly collapses:

```haskell
sad :: (BitWidth a ~ BitWidth b) => a -> State b ()
sad = putState @BitWidth @Int8 @Char
```

Given we have allowed `BitWidth` to stand unsaturated, we allow the type checker to substitute `BitWidth` for f. Once substituted, the type checker would derive that `BitWidth Int8` $\sim$ `BitWidth Char`. At this point, rather importantly, the type checker should happily decompose this constraint and derive `Int8` $\sim$ `Char`, which would be especially unfortunate.

Assumptions around the *matchability* of a type-level function are precisely what allow the type checker to arrive at this faulty conclusion (Eisenberg 2016; Kiss 2018; Chen 2024). A type-level function is matchable if (and only if) it is injective and generative, and assuming f to be matchable allows us to reverse-engineer $a \sim b$ from $f\ a \sim f\ b$. While unsaturated type families are absent, and f can only refer to matchable type-level functions, GHC's overly optimistic assumptions hold. However, to relieve the tension of this delicately balanced arrangement, we need to find an alternative way of tracking matchability.

A recently proposed solution relies on encoding the information in the kind of a type-level function (Kiss 2018). In this doctrine, we refer to a type-level function as matchable if the kind of the function is $* \rightarrow_M *$, and unmatchable if the kind of the function is $* \rightarrow_U *$. On the surface, this solution seems rather pleasant. The type checker is able to refer to the kind of the type-level function to determine whether it is sensible to choose to decompose a constraint. In this specific encoding of the information, however, problems arise.

To retain backward compatibility, the scheme must determine the kinds of unannotated functions. For top-level definitions of type-level functions, this inference is simple. Generally, data constructors are matchable and type families are not. However, difficulties quickly arise with various edge cases. Consider an example with higher-ranked types.

```
withFix :: (forall f. f (Fix f) -> r) -> r
withFix = ...
```

Here, f is a type-level function with an ambiguous matchability. While it seems sensible to then render the matchability of f a *type-level variable* that we later unify depending on the choice of f, GHC interferes. Under the current design of the compiler, GHC demands separate compilation of modules and enforces that all types exposed in the API of a module must be *principal types*. A principal type, quite unfortunately, is precisely a type without type-level variables.

In this specific case, we resort to defaulting the matchability of f to be unmatchable where it remains unknown. While it may seem tempting to resort to defaulting all ambiguous matchability to the conservative case, previous code directly relies on assumptions of matchability. Instead, in a rather fragile compromise, there are fourteen *defaulting rules* and an extra, separate, last-ditch fallback that attempt to cover the various edge cases that arise (Chen 2024). While tempting, then, to encode information in the kind of a type-level function, the tension that arises suggests a more natural resting place for matchability to potentially be elsewhere.

Decomposition of a constraint is a form of coercion and, as with any coercion, requires evidence that this coercion is sound. In GHC, the `Coercible` constraint often deals with providing the evidence for other kinds of coercions.

```
coerce :: Coercible a b => a -> b
coerce = ...
```

To switch between two *representationally equal* types, we explicitly coerce the type using `Coerce` which then demands that `Coercible a b` is true. Uniquely, the type checker is both the demander and the resolver of this constraint, due to the fact that `Coercible` is effectively closed to the user. The type checker is usefully aware of what types are representationally equal and relies on this knowledge to determine whether it can resolve the coercion.

Similarly, we find an elegant solution to the problem of encoding matchability through a `Matchable` constraint alongside a hidden modification to the kind arrow.

```
decompose :: Matchable f => (f a :~: f b) -> (a :~: b)
```

The type checker, when triggering a decomposition of a constraint of the form $f\ a \sim f\ b$, requires coercion evidence that f is matchable. Ensuring that `Matchable` is closed, the compiler inserts a proof of matchability for data constructors and newtypes but not type families. This is, in essence, a simple method of providing such coercion evidence to GHC and side-steps the complications that arise when lifting such evidence into the kind of a type-level function. Importantly, this technique has precedent in both the `Coercion` and `Typeable` constraints. In later chapters, we explore the theoretical and practical underpinnings of this approach.

2.3 Deciding Termination

Dependent Haskell unifies term-level computation with type-level computation, rather quickly introducing theoretical Turing completeness (Eisenberg 2016; Gundry 2013). As is periodically rediscovered, Turing complete models of computation render the halting problem undecidable (Berthelette, Brassard, and Coiteux-Roy 2024). This has made the decidability of a Dependent Haskell dubious.

Literature with respect to Dependent Haskell largely ignores this concern, accepting it as a pragmatic compromise (Eisenberg and Weirich 2012). This compromise, however, is quite unfortunate. Other dependently typed languages, like Agda, avoid such concerns by forcing all programs to be decidable through *size annotations* on types (Abel 2010). Retrofitting such a mechanism into Haskell would gravely impair the language. The inability for a programmer to make her programs non-terminating where desirable would render applications like web servers and operating systems both impossible to build, and no longer backwards compatible.

Agda is able to nonetheless prove properties on these programs by modelling them as *coinductive* processes and similarly demanding the *productivity*[1] of these processes (ABEL and PIENTKA 2016). This would, quite unsurprisingly, not be a pragmatic choice for a Dependent Haskell. Agda introduces a large breadth of syntax, annotations and restrictions to support such an approach which nonetheless heavily impedes the construction of useful programs.

An alternative approach is necessary where we might hope that a Dependent Haskell would be decidable: an approach that is expressive and largely allows the unimpeded construction of useful programs.

[1] Coinductive productivity is the categorical dual to termination. Where termination describes that a program must terminate, productivity demands that a program must always make progress through an infinite stream.

2.3.1 Sized Types

A method of guaranteeing the termination of a program is by proving the existence of a bounded and monotonically decreasing structural variant on the program. Put simply, given the established variant is monotonically decreasing and is lower bounded, we must eventually reach the lower bound and terminate. Sized types, a technique largely embraced by other dependently typed languages, offer a method of establishing such a variant (Abel 2010).

By annotating the size (and rather importantly, the size change) of types within recursive programs, we intend to establish that the size change of all recursive calls is either negative or corresponds to the base case of the type. Where these size changes are negative, we may describe the size of the type as monotonically decreasing and therefore enforcing that all recursive calls step closer to the base case.

```haskell
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

height :: Tree a -> Int
height Leaf           = 0
height (Node l _ r)   = 1 + max (height l) (height r)
```

In the example above, `height` is provably terminating under this scheme. The function recurses on its first (and only) argument, which structurally decreases at each step: the base case `Leaf` provides the lower bound structure, while the decomposition of `Node` into its subtrees ensures progress towards a `Leaf`.

Notably, many recursion schemes, such as catamorphisms and paramorphisms, are terminating for similar structural reasons (Meijer, Fokkinga, and Paterson 1991; Yang and Wu 2022). This offers languages like Agda an expressive suite of tools to express the computation of proofs. The termination of such proof checking is non-negotiable for a proof assistant, otherwise the logic is rendered unsound (Coquand and Huet 1988). Haskell, however, does very little to prevent programs from running forever and has rather embraced partiality.

Retrofitting Haskell to an annotated sized types based approach is infeasible, not without grievously impairing the language. Despite this, it is desirable to determine a scheme that would recover decidability while retaining the majority of the present expressivity of the language. Such a scheme would retain the logical soundness of the type system for a Dependent Haskell and eliminate the possibility the type checker might run forever.

2.3.2 System F & PCF

System F is a variation of the simply typed lambda calculus (STLC) that formalises parametric polymorphism (Girard 1972; Reynolds 1974) and is the precursor language to System FC, the theoretical underpinning of the Haskell language. Not unlike STLC, a well-typed System F program is strongly normalising and therefore always terminating.

As System FC enjoys a close relationship with System F (Weirich, Hsu, and Eisenberg 2013), System F is a particularly interesting subject of study with regards to modelling the computation of Haskell. The syntax, semantics and typing rules of System F are on the whole rather simple:

Syntax:

$$\begin{array}{rrcl} \text{Types} & \tau, \sigma & ::= & X \mid \tau \to \sigma \mid \forall X.\tau \\ \text{Terms} & e & ::= & x \mid \lambda x{:}\tau.e \mid e_1\, e_2 \mid \Lambda X.e \mid e\, \tau \\ \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \end{array}$$

Typing Rules:

$$\textsc{Var}\ \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\textsc{TAbs}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda X.e : \forall X.\tau} \qquad \textsc{TApp}\ \frac{\Gamma \vdash e : \forall X.\tau_1 \qquad \Gamma \vdash \tau_2 \text{ type}}{\Gamma \vdash e\, \tau_2 : \tau_1[X \mapsto \tau_2]}$$

Figure 2.1: System F: Syntax and Typing Rules

STLC, and therefore System F, with the introduction of a `Fix` combinator immediately renders the language undecidable. Two particularly apt examples of such result are the undecidability of Programmable Computable Functions (PCF) (Loader 2001; Plotkin 1977), and the undecidability of System FC (Weirich, Hsu, and Eisenberg 2013). System FC introduces recursive let bindings, enabling recursive computation, the basis for the undecidability of Haskell, though the reader is referred to the System FC technical report for a full formalisation of the language.[2]

Syntax:

$$\begin{array}{rrcl} \text{Types} & \tau & ::= & \mathtt{nat} \mid \tau \to \tau \\ \text{Terms} & M & ::= & x \mid \lambda x{:}\tau.M \mid M\, N \\ & & & \mid \mathtt{zero} \mid \mathtt{succ} \mid \mathtt{pred} \mid \mathtt{iszero} \\ & & & \mid \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ P \\ & & & \mid \mathtt{fix}_{(\tau)}\ M \\ \text{Values} & V & ::= & \lambda x{:}\tau.M \mid \mathtt{zero} \mid \mathtt{succ}\ V \end{array}$$

Figure 2.2: The Syntax of PCF

The syntax for PCF is also rather simple, and introduces a fix point combinator `Y` alongside other constants to a base syntax that is effectively an extension to STLC. PCF is provably

[2] The full technical report can be found at the following link:
https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1015&context=compsci_pubs

undecidable, including under the restriction that the cardinality of the single base type is finite, and even merely a boolean (Loader 2001; Plotkin 1977).

The presence of unguarded recursion and thus a fix point combinator is particularly important for a Dependent Haskell, and so far the introduction of fix point combinators into either of these models of computation quickly loses decidability. To recover decidability, yet maintain the majority of the expressive semantics of PCF, would be necessary to underpin a formalised model of computation for a Dependent Haskell.

2.3.3 Whistles

A whistle, in the context of supercompilation (Sørensen, Glück, and N. Jones 1996; Leuschel 2002), is often defined as a set of program traces W. Where a trace t is a member of W, we say the trace is admissible with respect to W, and vice versa. The decidability of finitary first-order term rewriting systems relies on the presence of a whistle that is capable of determining if a program trace is considered dangerous (i.e. non-terminating) or safe (Leuschel 2002; Dershowitz 1987). First-order term rewriting systems are defined with the following syntax:

Syntax:

$$\begin{aligned} \text{Signature} \quad & \Sigma = \{f_1, f_2, \dots\} \\ \text{Variables} \quad & \mathcal{X} = \{x, y, z, \dots\} \\ \text{Terms} \quad & \mathcal{T}(\Sigma, \mathcal{X}) ::= x \mid f(t_1, \dots, t_n) \text{ where } f \in \Sigma_n \end{aligned}$$

Rewrite Rules:

$$\mathcal{R} = \{l_i \to r_i \mid l_i, r_i \in \mathcal{T}(\Sigma, \mathcal{X}), \mathrm{Var}(r_i) \subseteq \mathrm{Var}(l_i)\}$$

Figure 2.3: First-Order Term Rewriting Systems

The construction of a whistle for finitary first-order term rewriting systems are based on Kruskal's tree theorem; an infinite trace of finite trees constructed from a finite set of symbols must have a pair of trees that self-embed under homeomorphic embedding(Leuschel 2002; Kruskal 1960; Dershowitz 1993). Where a term rewriting system is first-order and finite, we may construct a whistle of all finite traces that do not contain a self-embedding pair. The decidability of a first order term-rewriting system is therefore decidable in an online manner by maintaining that the trace of the program at each state must be admissible with respect to the whistle.

Interestingly, if all term rewriting rules are oriented such that the right-hand side is always smaller than the left-hand side under the well-quasi order defined by homeomorphic embedding, we may immediately statically determine the termination of the program (Dershowitz 1987). Such a set of rewrite rules effectively constitute a user-provided proof of termination of the program and avoid the necessity of maintaining a whistle to decide the program in an online manner. However, the ability to determine the termination of a program, albeit in an online manner, irrespective of rule orientation is particularly useful.

The use of whistles for first-order termination checking in an online manner suggests a way forward for higher-order languages. Rather than demand proof of strong normalisation through syntactic size-change termination analysis such as Sized Types, the monitoring of a program through a whistle allows for the decidability of the termination of the term rewriting system. To this end, Chapter 5 explores adapting this approach to a restricted polymorphic lambda calculus with a fix-point combinator, achieving decidability through a similar mechanism.

Chapter 3

Theory of Unsaturated Type Families

”Simplicity is not an end in art, but we usually arrive at simplicity as we approach the true sense of things.”
—Constantin Brâncuși

Matchability encoded as part of the arrow has so far been the accepted solution to differentiating between matchable and unmatchable type-level functions (Eisenberg 2016; Kiss 2018; Chen 2024). However, practical attempts at bringing this feature to GHC have found limitations in the form of numerous defaulting rules with open questions about the defaulting behaviour of certain language features (Chen 2024).

In this chapter, we consider the shortcomings of previous attempts and propose two sound approaches to support unsaturated type families. First, we consider a novel approach of encoding the matchability of a type-level function as a class predicate and illustrate the mechanism by which the soundness of the type system is preserved. Later, we analyse the current arrow annotation approach and show defaulting rules may be vastly simplified.

3.1 The Problem with Kind Arrows

Today, Haskell makes no differentiation between the kind of matchable and unmatchable type-level functions; both `Maybe :: * → *` and `Id :: * → *` have the same kind. To support an implementation of unsaturated type families within Haskell, the matchability of a type-level function must be considered when executing certain decomposition rules in the constraint solver.

Early proposals described splitting the arrow into two arrows, namely $(\to)$ and $(\twoheadrightarrow)$, describing matchable and unmatchable type-level functions respectively. Through this separation, the constraint solver is able to predicate decomposition that relies on matchability by the presence of a $(\to)$ arrow rather than a $(\twoheadrightarrow)$ arrow.

To illustrate why this distinction matters, we must look to the constraint solver. The decomposition of a constraint `f a ~ f b` to `a ~ b` is only valid when `f` is injective and generative. In Haskell, data types like `Maybe` meet this criteria due to their structural nature but type families do not.

Whilst this approach would solve the information problem for the constraint solver, it produces a new issue: type class definitions over type-level functions cannot consider both matchable and unmatchable functions simultaneously. To illustrate, consider the type class for `Functor`:

```haskell
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

The argument `f :: * → *` uses the $(\rightarrow)$ arrow and thus cannot be used to define a `Functor` instance for type families which are unmatchable. This creates an immediate problem: to write a Functor instance for any type family, we must define a separate `Functor'` instance concerning unmatchable types even though using the original class might be perfectly sensible.

```haskell
class Functor' (f :: * ->> *) where
    fmap' :: (a -> b) -> f a -> f b
```

It becomes rapidly apparent that this results in excessive code duplication. Library authors would need to choose between the construct supporting data constructors or support type families, but not both. Likewise, users would need to remember which version to use where. This would be rather unfortunate.

A solution in the form of considering the $(\rightarrow)$ arrow as a subtype of the $(\twoheadrightarrow)$ arrow would eliminate the need for code duplication but at the cost of weakened type inference (Kiss 2018). Kiss aptly provides the following example.

```haskell
instance Functor' Id where
    fmap' :: (a -> b) -> Id a -> Id b
    fmap' f x = f x

eleven :: Int
-- eleven = fmap' (+1) 10
eleven = fmap' @Id (+1) 10
```

The solver is able to infer that `eleven` has both the type `Int` and also, when the usage of `fmap` is considered, the type `f a` for some `f` and `a`. The solver must unify the two forms and find some valid substitution for `f` and `a` such that `f a ~ Int`. Through the arguments provided to `fmap`, it is clear to the solver that `a ~ Int`. However, it is not so easy to determine for what specific `f` we have the equality `f Int ~ Int`.

A refinement was proposed in the form of making arrows matchability-polymorphic instead of relying on the sub-typing relationship (Kiss 2018). In short, Haskell would remain with a single arrow $(\rightarrow_m)$ where m is a polymorphic matchability parameter. One could then define $(\rightarrow) = (\rightarrow_M)$ and $(\twoheadrightarrow) = (\rightarrow_U)$ where M and U refer to 'Matchable' and 'Unmatchable' respectively. However, this does not avoid the introduction of new problems either.

A recent attempt at introducing a matchability-polymorphic arrow into GHC encountered the need to introduce 15 separate defaulting rules for language constructs. Despite the numerous defaulting rules, the proposal remained with open questions regarding the treatment of standalone kind signatures and higher-ranked types (Chen 2024). The need for defaulting rules is not a new consideration; the work on encoding the multiplicity of types encountered the same need to introduce a (single) defaulting rule (Spiwack 2018). Unfortunately, any design of a new language feature which risks introducing unresolved metavariables must take into consideration defaulting rules, due to separate compilation.

The attempt at stratifying arrows into different matchability has thus encountered several issues. An alternative approach that side-steps the need for such defaulting rules is by considering matchability as a candidate for a new built-in type class constraint. The motivation is simple: the matchability of a type-level function is consistent during the entire lifetime of a program. Importantly, this is unlike other properties that have necessitated encoding in the arrow; the intended multiplicity of an argument of, say, `Int` is call-site dependent.

3.2 Matchability as a Class Predicate

Introducing a new `Matchable` type class is not unprecedented. Both `Coercible` and `Typeable` exist as closed, built-in type classes that encode various stable properties of types. In the case of `Matchable`, GHC would consider all matchable type-level functions to effectively have an instance of `Matchable`. This formulation is simple but poses questions regarding backwards compatibility. In present Haskell code, the following function `id'` type checks:

```haskell
id' :: (f a ~ f b) => a -> b
id' = id
```

The reason the unusual signature for `id'` is accepted is precisely due to the assumption of the matchability of `f`. In short, `f` is assumed matchable and therefore the solver finds that `f a` $\sim$ `f b` $\implies$ `a` $\sim$ `b`. The concern is now that it may become necessary to add the explicit constraint `Matchable f` into the signature under the new scheme:

```haskell
id' :: (f a ~ f b, Matchable f) => a -> b
id' = id
```

This is, however, not necessary. Where the solver finds that it requires decomposition to derive an equality (such as `a` $\sim$ `b`), the solver may simply emit that it now wants the constraint `Matchable f` to be fulfilled. This emission is a form of constraint floating and effectively constitutes an adequate inference of the constraint `Matchable f` where the user relied on but did not explicitly state it (Vytiniotis et al. 2011).

After emission, the wanted `Matchable f` constraint is solved for the individual case on an instantiation-per-instantiation basis. Where an appropriate concrete type-level function is provided for `f`, the solver will check the nature of the function and discharge the constraint if it determines the function injective and generative. As such, the evidence for `Matchable f` is generated on-the-fly and on a per-needed basis.

A final concern rests with the apartness of applied type families (Kiss 2018) (Chen 2024). In

Haskell, the presence of type families in type class instance heads is restricted due to instance coherence concerns. In short, the unrestricted presence of type families would render instance resolution undecidable. Determining the apartness of types is the responsibility of the unifier rather than the type checker, which poses a problem for the evidence-based approach.

A benefit of the kind-arrow based approach is that the unifier, which does not have access to the type checker's environment, is able to directly reference the kind of a metavariable in deciding apartness. Thus, the approach taken by Kiss has been to unify under a `Maybe Apart` decision where the kind of the type variable is known to be unmatchable (Kiss 2018).

One difficulty with the unifier is how to interpret arrows that do not have a known matchability. The conservative and sound decision would be to assume the most lax default; an ambiguous arrow is an unmatchable arrow. This would, however, break backwards compatibility given current assumptions in GHC that all higher-kinded free type variables are matchable.

This is predominantly the source of the numerous defaulting rules in earlier approaches. In the evidence-based approach, we opt to respect the assumption that all higher-kinded free types are matchable and allow the user to specify otherwise via an `Unmatchable` class constraint.

3.3 Reintroducing Kind Arrows

Where matchability only concerns the constraint solver, the evidence-based design of floating a `Matchable` constraint into the type signature would be fully sufficient to ensure the soundness of the introduction of unsaturated type families. However, architectural decisions within GHC pose limitations. As part of the GHC implementation, lightweight variable unification is done by the unifier and not the constraint solver.

There are two unifiers: a pure unifier and a unifier that is capable of emitting coercions. Neither of these unifiers have full access to the constraint solver environment. Furthermore, the pure unifier is sometimes utilised outside of the type checking phase. The only information the unifier has to base unification decisions on is the types and kinds it is provided with.

This is particularly pertinent for the theory of unsaturated type families. Simply put, the unifier must know how to answer the question of whether `f a` $\sim$ `Int`. Where unsaturated type families are not first-class, `f` may only be substituted for a matchable type-level function. Therefore the unifier is able to conclude that `f a` and `Int` are `Surely Apart`, as `f a` does not structurally decompose to `Int`.

However, when we introduce unsaturated type families, this reasoning breaks down. The type `f a` may reduce to `Int` after reduction, or it may reduce to some other type. Therefore no decision on unification can be made at the present moment and unification must be deferred. The unifier is able to specify that the types are `Maybe Apart` when unifying `F a` $\sim$ `Int` to represent this deferral where `F` is a concrete type family.

Therefore it may be tempting to imagine that a sensible conclusion for the reduction of `f a` $\sim$ `Int` is to also conservatively conclude that the two are always `Maybe Apart`. As such, the unifier would systematically defer all such unification decisions until types are substituted for the free variables and reduction can be performed. This would ensure soundness by avoiding the case that two types are prematurely considered apart where they may reduce to equal types later. However, like the conservative case for kind defaulting, conservative behaviour crucially breaks backwards compatibility, a key condition for a successful theoretical basis for unsaturated type families.[1]

Likewise, assuming matchability within the unifier at all times breaks instance head resolution coherence. The compiler would always determine `f a ~ Int` to be surely apart, even if `f a` were to reduce to `Int` in the future. To illustrate the concern, consider the following example:

```
class Outputable a where
    showItem :: a -> IO ()

instance Outputable Int where
    showItem a = print a

instance Outputable (f a) where
    showItem a = print "Unsure"
```

The type class `Outputable` has instances for both `Int` and `f a`. Where unsaturated type families are not present, these instances do not overlap. However, where unsaturated type families are present, the instances do overlap. In the latter case, depending on the state of the reduction of the type, instance lookup may produce two different behaviours for the same type. Different behaviours for the same type is a rather big problem for the coherence of program behaviour within GHC; it is vital that GHC always picks the same instance for the type.

Therefore it becomes necessary to preload the unifier with this information in the form of encoding matchability in the kind arrow. Contrary to previous implementations, this approach hides the annotation for the kind arrow, leaving it as a hidden phantom marker. This requires, however, that the correct annotations are inferred. In other words, there must exist a bridge between the class predicate world and the kind arrow world. To achieve this, we unify the two worlds by demanding that on discharge of the matchability constraint for some higher-kinded type variable `f :: k1` $\rightarrow_M$ `k2`, we emit a new given, namely `[G] m ~ 'M` where `'M` is the annotation for a matchable kind arrow.

Given the kind annotation is now predicated on the class constraint for the type variable, all type variables must now be guarded. This requires the desugaring of a signature to infer `Matchable` constraints on unguarded type variables, reflecting the backwards-compatible assumption that unannotated type variables are matchable. To illustrate this process, consider the following signature:

```
foo :: f a -> f b
```

Under such a desugaring scheme, the signature would inferred to be:

```
foo :: Matchable f => f a -> f b
```

While new signatures that are already annotated with `Unmatchable` are left alone, as follows:

[1]In GHC, reduction is performed through coercion axioms for type families and newtypes, but data types are treated structurally. The unification of this representation would likely render this implementation dramatically simpler.

```
-- Remains unmodified
bar :: Unmatchable f => f a -> f b
```

Hence, we simplify the machinery drastically by forcing evidence to be provided on instantiation and once evidence is provided, the relevant correct matchability is assigned depending on the instantiation.

3.4 The Type System

The two above approaches both intend to solve one problem: encode the evidence of matchability in some type-level construct and ensure the type system uses this evidence to correctly predicate constraint solver and unifier behaviour. Any theory of unsaturated type families must achieve three things: respect the backwards compatible assumption that unannotated type variables are matchable, infer the matchability of concrete type-level functions and construct a suitable evidence chain to pass evidence through substitutions.

Both the kind arrow annotation approach and the class evidence based approach achieve the construction of such an evidence chain. Type equality in System FC relies on decomposition rules that are only sound for matchable type functions. To illustrate, consider the equality `Maybe Int` $\sim$ `Maybe Bool`. Given the structural nature of `Maybe`, we can decompose the above equality into `Int` $\sim$ `Bool`. The decomposed equality is false and therefore we are able to reject the full equality `Maybe Int` $\sim$ `Maybe Bool` on this basis.

Where the equality is rather `F Int` $\sim$ `F Bool`, where `F` is some type family, we cannot structurally decompose the equality as above. Despite the fact that `Int` $\sim$ `Bool` is not true, `F` represents some arbitrary computation that may transform `Int` and `Bool` to the same type, therefore rendering `F Int` and `F Bool` equal types. The equality of this constraint is therefore determined by reduction rather than structural decomposition.

However, due to the fact that only matchable type functions are allowed to appear unsaturated, the constraint solver will apply decomposition to any constraint of the form `f a` $\sim$ `g b` for some free type variables `f` and `g`. As only matchable type-level functions are allowed to appear unsaturated, `f` and `g` may only be substituted for matchable type functions. In lifting this restriction, the typing rules must be modified to track the matchability of type-level functions. In the kind-arrow approach, this is achieved by annotating arrows with matchability markers and then predicating the behaviour of the constraint solver on the matchability of the arrow.

In System FC, type equalities are witnessed by coercion terms (Weirich, Hsu, and Eisenberg 2013). These coercions are proofs of equality provided to the solver, allowing the solver to use the available proof objects to determine whether two types are truly equal. Given coercion terms are only useful in the type checking stage and are not relevant for runtime behaviour, they are erased alongside other type information during compilation. To render the type system sound in the presence of unsaturated type families, the coercion formation rules must be modified to only decompose coercions where matchability is guaranteed. To formally illustrate, consider the following coercion formation rules (Kiss 2018):

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

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \textbf{refl}\ \tau : \tau \sim \tau}\text{Co\_Refl}$$

$$\frac{\Gamma \vdash \gamma : \sigma \sim \tau}{\Gamma \vdash \textbf{sym}\ \gamma : \tau \sim \sigma}\text{Co\_Sym}$$

$$\frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \tau_2 \quad \Gamma \vdash \gamma_2 : \tau_2 \sim \tau_3}{\Gamma \vdash \textbf{trans}\ \gamma_1\ \gamma_2 : \tau_1 \sim \tau_3}\text{Co\_Trans}$$

$$\frac{\vdash \Gamma \quad c : \tau \sim \sigma \in \Gamma}{\Gamma \vdash c : \tau \sim \sigma}\text{Co\_Var}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash \gamma : \tau_1\ @@^{M}\ \tau_2 \sim \sigma_1\ @@^{M}\ \sigma_2 \\ \Gamma \vdash \tau_1 : \kappa \quad \Gamma \vdash \sigma_1 : \kappa\end{array}}{\Gamma \vdash \textbf{left}\ \gamma : \tau_1 \sim \sigma_1}\text{Co\_Left}$$

$$\frac{\begin{array}{c}\Gamma \vdash \gamma : \tau_1\ @@^{M}\ \tau_2 \sim \sigma_1\ @@^{M}\ \sigma_2 \\ \Gamma \vdash \tau_1 : \kappa \quad \Gamma \vdash \sigma_1 : \kappa\end{array}}{\Gamma \vdash \textbf{right}\ \gamma : \tau_2 \sim \sigma_2}\text{Co\_Right}$$

$$\frac{C : [a : \kappa].\sigma_1 \sim \sigma_2 \quad \vdash \Gamma \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash C(\overline{\tau}) : \sigma_1[\overline{\tau}/a] \sim \sigma_2[\overline{\tau}/a]}\text{Co\_Axiom}$$

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

The `Co_Left` and `Co_Right` rules represented the decomposition discussed. In the premise for each rule, there is an obligation that the functions τ_1 and σ_1 are matchable through the M subscript. As such, the type checker only applies this rule in a sound manner.

In the class constraint approach, evidence for coercion formation takes a different form. Instead of annotations on arrows, `Matchable` constraints witness matchability. These two forms of evidence are equivalent; `Matchable f` simply denotes that `f` is matchable, similarly to how `f :: k1` $\rightarrow_M$ `k2` likewise denotes this. The correspondence is made explicit during constraint solving, through the emission of a given equality `[G] m ~ 'M` on discharge, to ensure the unifier may base decisions on evidence accessible at time of unification. Therefore, to maintain the backwards compatibility invariant mentioned earlier, the class constraint approach must elaborate type signatures to infer class constraints where needed. The present expression typing rules for System FC are as follows:

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

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

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

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

$$\frac{\Gamma \vdash e : (\sigma_1 \sim \sigma_2) \Rightarrow \tau \quad \Gamma \vdash \gamma : \sigma_1 \sim \sigma_2}{\Gamma \vdash e\, \gamma : \tau}\text{E\_CApp} \qquad \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash e \rhd \gamma : \tau_2}\text{E\_Cast}$$

These rules make no present assumptions about matchability and instead treat all arrows uniformly. To ensure that arrow kinds are appropriately annotated and evidence is appropriately guarded, we extend expression typing rules with the following two elaboration rules that run subsequently after typing. [2]

$$\frac{\Gamma \vdash e : \forall (f : \kappa_1 \to \kappa_2).\, \sigma}{\Gamma \vdash e : \forall (f : \kappa_1 \to \kappa_2).\, \sigma \ \rightsquigarrow\ \forall (f : \kappa_1 \to^{m} \kappa_2).\, \text{Matchable } f \Rightarrow \sigma}$$

$$\frac{\Gamma \vdash e : \forall (f : \kappa_1 \to \kappa_2).\, \text{Unmatchable } f \Rightarrow \sigma}{\Gamma \vdash e : \forall (f : \kappa_1 \to \kappa_2).\, \sigma \ \rightsquigarrow\ \forall (f : \kappa_1 \to^{u} \kappa_2).\, \text{Unmatchable } f \Rightarrow \sigma}$$

These rules apply during the processing of type signatures and are only applied after the expression typing has completed. The first rule maintains the backwards compatibility invariant while the second preserves the explicit `Unmatchable` already present in the signature.

3.5 Fixing Kind Arrows

Previous approaches fell short due to the numerous defaulting rules necessary to encode matchability in the kind. Early approaches took a provenance-based matchability defaulting, where the matchability annotation on a kind arrow was determined by the syntactic context. The final attempt of such approaches presents 15 defaulting rules, yet still ends with open questions regarding standalone kind signatures and higher-ranked types, suggesting a fundamentally flawed approach. The aforementioned defaulting rules are enumerated in Table 3.1 (Chen 2024).

The provenance based approach tags each type constructor kind; type families kinds are tagged as "defined in a type family," data type kinds as "defined in a data declaration," and so on. The approach then propagates these tags through the type system, using the tags to determine the matchability defaults in each of the fifteen contexts enumerated.

[2]The precise formalisation for System FC, "System FC, as implemented in GHC" (Eisenberg 2015), does not provide a treatment for type classes. However, for completeness, type classes are included in the elaboration rules and should be considered a constraint on the signature.

Context	**Defaults to**	**Hard Rule?**
Type constructor/class/data family kind	Matchable	Yes
Promoted data constructor kind	Matchable	Yes
Application in type patterns	Matchable	Yes
Type family/synonym kind	Unmatchable	Yes
Data constructor/function type	Unmatchable	Yes
Pattern synonym type	Unmatchable	Yes
Kind inside function signature	Matchable if ext. off	No
Kind in declaration & type synonym	Matchable if ext. off	No
Other type application	Matchable	No
Type variable with arrow kind	Matchable	No
Kind in type class parameter	Matchable	No
Other ambiguous kind arrow	Matchable	No
Instance on arrow ($\rightarrow$)	Unmatchable	No
Ambiguous arrow type	Unmatchable	No
Fall-back for constraint solver	Matchable	N/A

Table 3.1: Matchability Defaulting Rules

The proliferation of rules suggests a fundamental problem within the chosen architecture. The system attempts to infer semantic properties through syntactic provenance. Such inference creates an indirect mapping, where the definition site informs provenance, which informs the defaulting rule, which informs the final annotation. Indeed, this is a rather elaborate approach for asking whether a type is both injective and generative.

The structural analysis employed in the class constraint approach suggests a simpler alternative. When discharging `Matchable` constraints, a solution is determined through structural inspection of the type guarded by the class. GHC provides two built-in predicates, `isInjectiveTyCon` and `isGenerativeTyCon`, that power this check.[3] Rather than inferring these properties through provenance, it is possible to query them directly. Therefore, this thesis proposes the following solution to the open problem of kind arrow defaulting: where a kind arrow requires annotation, simply determine the annotation by structural inspection of the type constructor that will inhabit this kind.

This approach offers several theoretical and practical advantages. Firstly, matchability annotations reflect the known properties of types within GHC rather than definition location. Secondly, every type constructor can be analysed in a coherent and complete way, eliminating various gaps in the above approach. Thirdly, a single uniform rule drastically simplifies the architectural requirements necessary to determine the annotation of a type constructor. Finally, new syntactic features within Haskell will not require new defaulting rules.

The structural defaulting approach as described demonstrates the complexity of previous attempts was unnecessary. By synchronising the defaulting strategy with underlying semantic properties known to the compiler, a simpler and more principled design is achieved; matchability is fundamentally about type semantics, not syntax or provenance. This unifies defaulting under a simple rule: query whether the type is injective and generative.

[3]The source for GHC can be found at: https://gitlab.haskell.org/ghc/ghc

Chapter 4

Practice of Unsaturated Type Families

"One must learn by doing the thing; for though you think you know it,
you have no certainty until you try."
—Sophocles

This chapter presents the practical implications of two type-theoretically sound extensions to GHC, each designed to implement unsaturated type families. The first of these implements the class-based approach and explores the implementation of the framework discussed in Chapter 3. The second of these extensions simplifies the considerable number of defaulting rules into a single unified structural check, closing open questions left by previous literature.

Where previous attempts have focused primarily on the theory and practice of the kind-arrow approach (Kiss 2018; Chen 2024; Eisenberg 2016), this chapter explores the implementation of both approaches. To this end, this chapter demonstrates that the feasibility of an implementation within GHC relies not only on theoretical soundness but also on interactions with latent architectural assumptions within the compiler pipeline.

4.1 The Solver

The constraint solver is a key component of the compiler frontend that must soundly support unsaturated type families, given the role of the solver in coercion formation. Where the constraint solver relied on type decomposition of the form $\texttt{f a} \sim \texttt{f b} \implies \texttt{a} \sim \texttt{b}$, the solver implicitly assumed both the injectivity and generativity of f. The introduction of the Matchable class predicate was devised as a method guarding this assumption; only type-level functions that were indeed matchable would have an instance for the Matchable class.

`Coercible` and `Typeable` as class predicates act as precedent for this approach. Both classes represent an assertion that the types guarded by the class have certain properties and are built-in classes that enjoy special treatment. With regards to the `Matchable` class, the goal is simple: where type checking relied on the matchability of `f`, it may not be substituted for an unmatchable `f`.

Therefore, the implementation for this approach follows naturally from the theory described in Chapter 3. Despite the significant mechanical work necessary to admit any new implementation into GHC, we describe the more pertinent changes focused predominantly in three main areas: so-called ”wired-in” constructs, the constraint solver and the instance lookup system. We start with the wired-in constructs.

The introduction of the Matchable type class is defined as a built-in internal type within GHC, similarly to Coercible:

```
-- | Class for matchable type constructors. This allows the constraint
    solver
-- to decompose types like @f a ~ f b@ into @a ~ b@ when @f@ is known
    to be
-- matchable. A type constructor is matchable if it is injective and
    generative.

class Matchable (f :: k)
```

After registering `Matchable` as a built-in class, the class is now available for use within Haskell and therefore within the constraint solver. We next modify the constraint solver itself.

Within the constraint solver, the function `can_eq_nc` is responsible for determining the equality of two types through coercion formation and is part of the implementation for `OutsideIn(X)` within GHC. To implement the necessary changes within the solver logic, the following modification is employed:

```
can_eq_nc True _rdr_env _envs _ienvs ev NomEq ty1 _ ty2 _
  | Just (t1, s1) <- tcSplitAppTy_maybe ty1
  , Just (t2, s2) <- tcSplitAppTy_maybe ty2
  = do { matchable_class <- tcLookupClass matchableClassName
      ; unless ((t1 eqType t2 && s1 eqType s2) || isWanted ev) $ do {
          ; emitMatchableConstraint matchable_class ev t1
          ; emitMatchableConstraint matchable_class ev t2 }
      ; can_eq_app ev t1 s1 t2 s2 }
```

In short, the constraint solver now emits a Matchable constraint whenever it requires the decomposition of `f a ∼ f b ⟹ a ∼ b`. Where this constraint remains unresolved, as it always is for some free type variable `f`, GHC will simply generalise the wanted Matchable constraint emitted into the signature of the function being type checked (Vytiniotis et al. 2011). To illustrate, consider an example:

```
foo :: (f a ~ f b) => a -> b
foo = id
```

The function foo relies on the decomposition `f a ∼ f b ⟹ a ∼ b` to type check, as it must find `a ∼ b` to admit the implementation `foo = id`. The constraint solver, per the above change, will generalise the demand Matchable f into the signature:

```
foo :: (Matchable f, f a ~ f b) => a -> b
```

Hence, the matchability of f is guarded and soundness is preserved. The final consideration is with instance resolution for the `Matchable f` constraint. While automatically generating `instance Matchable` bindings for all matchable type-level functions is tempting, the simple approach is to find evidence on-the-fly. This approach is possible as Matchable represents the constraint that a type is injective and generative, a property we can determine simply by analysing the type within the Matchable constraint.

```
mkMatchableInst :: Class -> [Type] -> [PredType] -> TcM ClsInstResult
mkMatchableInst cls [ki, ty] theta = do
  return $ OneInst { cir_new_theta = theta
                       , cir_mk_ev = \_ -> EvExpr (mkCoreConApps (
   classDataCon cls) [Type ki, Type ty])
                       , cir_canonical = EvCanonical
                       , cir_what = BuiltinInstance }
mkMatchableInst _ _ _ = return NoInstance

matchMatchable :: Class -> [Type] -> TcM ClsInstResult
matchMatchable cls [ki, CastTy ty' _] = matchMatchable cls [ki, ty']

matchMatchable cls tys' = do
  case tys' of
    [ki, ty]
        | Just (tc, _) <- tcSplitTyConApp_maybe ty,
          isMatchableTyCon tc Nominal -> mkMatchableInst cls tys' []
    [ki, AppTy tyh _] -> mkMatchableInst cls tys' [mkClassPred cls [
   typeKind tyh, tyh]]
    _                 -> return $ NoInstance
```

This technique is similar to that used to solve `Typeable` instances. Alongside these three core changes, a number of less interesting changes are employed to ensure these components interact as expected. On this foundation, the constraint solver retains soundness and, at least from the perspective of the solver, we may relax the constraint that type families must appear saturated. However, due to latent architectural decisions in both separating type checking concerns over multiple components as well as the lack of uniformity within type-level function reduction, soundness must be preserved elsewhere.

4.2 The Unifier

4.2.1 The Apartness Problem

The constraint solver is not the only part of the compiler that is responsible for determining the equality of two types. Instead, GHC separates type equality checking over multiple components, including two unifiers.

The two unifiers differ only slightly in mechanics; they are both responsible for the unification of types. The first, often called the pure unifier, simply attempts to unify types based on the structure of the type and determines equality up to substitution. The latter unifier is able to utilise coercions to determine whether the two types are equal and produces a new coercion to act as evidence. After the new coercion is produced, it is emitted into the constraint solver environment to be used in the future.

The pure unifier, importantly, does not have access to information regarding class constraints residing in the constraint solver. This poses a problem for the coherence of unification. Consider the unification of two types: `f a` and `Int`. Under the assumption that only matchable type-level functions may appear unsaturated, we may be confident that `f a` $\sim$ `Int` will never unify even if `f` and `a` are unknown. This is for a simple reason: data constructors and newtypes are structural and do not represent arbitrary type-level computation. However, should `f` be able to represent an arbitrary type-level function, these assumptions must be reconsidered.

Where `f` may be substituted for a type family, the type family may or may not reduce to `Int` under some future input `a`. This level of caution must also be reflected in the unifier; the unifier must return a judgement that these two types are `Maybe Apart` as opposed to `Unifies` or `Surely Apart`. The Maybe Apart judgement allows the deferral of unification until `f a` is reduced once coercion axioms become available after substitution. The conservative solution then would be to have the unifier admit the judgement that `f a` and `Int` are indeed `Maybe Apart`. However, this immediately sacrifices backwards compatibility, as previously discussed.

Therefore, to preserve backwards compatibility, the assumption that `f` is by default matchable and only otherwise unmatchable if explicitly annotated must be encoded in a manner obtainable by the unifier. As the unifier can examine kinds, encoding matchability information in the kind yields a solution, assuming it is possible to avoid the numerous defaulting rules.

To encode matchability in the kind requires a modification to the internal representation of arrows. Firstly, the representation of a function type must admit a matchability flag:

```haskell
data Type
  = {- Other constructors omitted -}

  | FunTy      -- ^ FUN m m' t1 t2
     { ft_af   :: FunTyFlag

     , ft_mult :: Mult              -- Multiplicity
     , ft_mat  :: Mat               -- Matchability
     , ft_arg  :: Type              -- Argument type
     , ft_res  :: Type }            -- Result type

    {- Other constructors omitted -}
```

From there, the usual $(\rightarrow)$ arrow is modified to default to annotation `Matchable` while, for utility and ergonomics, a second $(-\#)$ unmatchable arrow is provided. A final matchability-polymorphic arrow is omitted from this presentation as it is simply an uninteresting third variation of the aforementioned two arrows.

```
type (->) = FUN 'Many 'M
type (-#) = FUN 'Many 'U
```

The above arrows stratify the arrow space into matchable and unmatchable arrows. Where necessary, in the following two approaches, the type checker may override the phantom marker in the arrow to reflect the true nature of the arrow. Concretely, in the case of signature elaboration, the compiler may override the arrow $(\rightarrow)$ and replace it with the arrow $(-\#)$ in the kind of a bounded type variable if the variable is guarded by an `Unmatchable` constraint. Likewise, in the defaulting rule unification approach, the compiler may override the usage of $(\rightarrow)$ arrows to describe the kind of concretely unmatchable type-level functions and simply replaces them with $(-\#)$.

4.2.2 Type Elaboration

A possible solution to avoid the numerous provenance-based defaulting rules discussed in earlier works is to fully elaborate all signatures, regardless of whether the solver demands matchability. That is to say, a second Unmatchable class may be introduced to mark the explicit annotation that `f` is unmatchable and, for every relevant signature in the program, all unguarded type variables receive a Matchable constraint as a default. Given the evidence discharge for Matchable is already predicated on a structural check of the type, we successfully encode the assumption that all bounded type variables are matchable by default.

To bridge the constraints with the kinds of the various bounded type variables, we simply add an annotation to the kind arrow according to the guard present. Then, when solved, Matchable f constraints may simply discharge the constraint that `m` $\sim$ `'Matchable` where `f :: k1` $\rightarrow_m$ `k2`. Likewise, Unmatchable f constraints may discharge `m` $\sim$ `'Unmatchable` on solution.

The algorithm to perform this elaboration requires coordinating across numerous loosely-coupled GHC components. With that in mind, we present a simplified version below.

```
-- Entry point
desugarMatchability :: Type -> Type
desugarMatchability ty =
  let (tvs, constraints, body) = decompose ty
      tvs' = inferMatchability tvs constraints
  in reconstruct tvs' constraints body

-- Decompose forall a b. theta => tau into components
decompose :: Type -> ([TyVar], [Constraint], Type)

-- Infer matchability for higher-kinded variables
inferMatchability :: [TyVar] -> [Constraint] -> [(TyVar, Matchability)]
inferMatchability tvs constrs =
  [ (tv, m) | tv <- tvs, let m = getMatchability tv constrs ]
  where
    getMatchability tv cs
      | tv `elem` constrainedVars cs =
          -- Annotated by the user
          lookupMatchability tv cs
      | isHigherKinded tv =
          -- Higher-kinded but unconstrained, default to a matchable
          Matchable
      | otherwise =
          -- Not higher-kinded and therefore no constraint
          NoConstraint

-- Reconstruct type
reconstruct :: [(TyVar, Matchability)] -> [Constraint] -> Type -> Type
```

Desugaring in this manner introduces a large number of class constraints in every signature. In Core, the underlying intermediate representation for Haskell based on System FC, class constraint evidence is represented as a so-called dictionary. Dictionaries will contain the set of methods defined by the class for use by the callee as well as any superclasses. For illustration, consider the following function addNums:

```
addNums :: Num a => a -> a -> a
addNums = (+)
```

During compilation, `addNums` is desugared from Haskell syntax to Core in a representation that looks roughly as follows:

```
addNums = /\a. \(d_Num :: Num a) (x :: a) (y :: a).
        -- rest of implementation
```

Importantly, there now exists a binder for the dictionary where there was a `Num a` constraint. Should Matchable be a non-trivial class with methods and superclasses, this would naturally be a significant concern. The runtime overhead of such desugaring would likely gravely impact performance. However, Matchable is a trivial dictionary and would simply be an empty boxed value that one might suspect should be optimised out. This assumption will prove to be incorrect. Whilst GHC does optimise out other empty unboxed values such as `void#`, it does not optimise out `Unit`-like data. A discussion on runtime considerations for this approach will be given in Section 4.3.

4.2.3 Maintaining an Evidence Chain

Where `Matchable` constraints are elaborated into every type signature, a consistent and implicit evidence chain is demanded. Under normal circumstances, the soundness of this evidence chain is maintained. However, rather notably, Haskell supports the unsound usage of `unsafeCoerce :: a → b` to forcibly coerce a term between two types. The incorrect usage of `unsafeCoerce` can cause undefined behaviour but nonetheless is wired-in as an escape hatch should it be necessary. This poses a problem for the evidence chain. Consider the following example:

```haskell
impossible :: forall f b. Matchable f =>
              f Int ->
              (forall b. Matchable b => b Int -> b Bool) ->
              f Bool
impossible x k = k corrupted
  where
    corrupted :: b Int
    corrupted = unsafeCoerce x
```

The unsafe coercion of `x` from the type `f Int` to `b Int` immediately disrupts the evidence chain. The reason is simple: despite the fact that `unsafeCoerce` did unsafely coerce between `f Int` and `b Int`, this coercion is not reproducible beyond the scope of `unsafeCoerce` itself. That is, the type checker cannot now rely on an equality between the two types in the future. Therefore, the evidence for `Matchable f` is not applicable for `Matchable b`, despite referring to the same runtime value. Indeed, this leaves the obligation `Matchable b` impossible to solve.

This problem admits a solution in the form of simply discharging Matchable obligations pertaining to type variables that remain unsolved. This effectively constitutes the admission that not only can `unsafeCoerce` coerce arbitrarily between types but, where necessary, it may also fabricate evidence regarding matchability to satisfy the type checker. Such fabrication of evidence, and the unsoundness of such, is not a concern of this thesis as `unsafeCoerce` is naturally already unsound.

4.2.4 Code Generation

Another problem that arises under this implementation is the ad-hoc code generation mechanism for various built-in constructs that now require `Matchable` dictionaries: namely Template Haskell machinery, Typeable machinery and the primitive `unsafeEqualityProof`. Despite the various constructs being defined in the compiler, the definitions in the library are stub definitions which the compiler will replace with custom-generated Core. Despite the definitions being constructed by the compiler, the signatures for the stubs are liable to be desugared in the manner described above. This indeed means that high-level code generation for these constructs must be altered to similarly fabricate evidence.

Evidence fabrication in this specific case, unlike `unsafeCoerce`, is actually sound. Before high-level code generation runs, the type checker will have validated that the evidence chain as described by the types is sound up to `unsafeCoerce`. Therefore, where the various built-in constructs demand a matchable dictionary, typical code generation would have provided evidence through the chain. Given evidence for `Matchable` is a trivial, empty dictionary, it would not necessarily matter whether an empty dictionary was handed-off through the evidence chain or through fabrication. The fabrication of such evidence to restore the chain is simply an artefact of the architectural decision to employ custom code generation.

We omit a large amount of the surrounding code generation logic to focus on the core parts of the particular changes required. Firstly, in the desugaring of Typeable related machinery, the signature has now been desugared to expect dictionaries for `SomeTypeRep`, `mkTrCon` and `mkTrApp`, to which the code generation must now respect:

```
-- ...
-- Due to SomeTypeRep :: forall k (a :: k). Matchable a => TypeRep a ->
    SomeTypeRep
 toSomeTypeRep :: Type -> EvTerm -> DsM CoreExpr
 toSomeTypeRep t ev = do
         rep <- getRep ev t
         return $ mkCoreConApps someTypeRepDataCon [Type (typeKind t),
   Type t, matchableEv matchableDataCon t, rep]

-- ...
-- Due to forall k1 k2 (a :: k1 -> k2) (b :: k1). (Matchable a,
   Matchable b)
--      => TypeRep a -> TypeRep b -> TypeRep (a b)
let expr =  mkApps (mkTyApps (Var mkTrApp) [ k1, k2, t1, t2 ])
        [ matchableEv matchableDataCon t1 , matchableEv
   matchableDataCon t2, e1, e2 ]

-- ...
-- Due to mkTrCon :: forall k (a :: k). Matchable a => TyCon -> TypeRep
    k -> TypeRep a
let expr = mkApps (Var mkTrCon) [ Type (typeKind ty)
 , Type ty
 , matchableEv matchableDataCon ty
 , tc_rep
 , kind_args ]
```

Secondly, where dictionaries are expected for the unsafeEqualityProof built-in primitives, they must also be provided:

```
 -- Due to unsafeEqualityProof :: forall k (a :: k) (b :: k). (
   Matchable a, Matchable b) => UnsafeEquality a b
 unsafe_equality k a b
   = ( mkTyApps (Var unsafe_equality_proof_id) [k,b,a] mkApps [mbdict,
   madict]
         , mkTyConApp unsafe_equality_tc [k,b,a]
         , mkNomEqPred a b
         )
         where
           madict = mkCoreConApps matchable_dcon [Type k, Type a]
           mbdict = mkCoreConApps matchable_dcon [Type k, Type b]
```

Thirdly, with regards to Template Haskell, necessary constructs are now desugared to fabricate new dictionaries:

```
-- ...
monadWrapper = mkWpEvApps [EvExpr $ mkCoreApps (Var monad_sel) [m_ty,
   Var matchable_ev, quote_var]] <.> mkWpTyApps [m_var]
```

```
-- ...
applyQuoteWrapper :: QuoteWrapper -> HsWrapper
applyQuoteWrapper (QuoteWrapper ev_var match_ev m_var)
  = mkWpEvVarApps [ev_var] <.> mkWpEvVarApps [match_ev] <.> mkWpTyApps [
    m_var]
```

To complete the process, the instance resolution mechanism is also changed to reflect the trivial discharge of `Unmatchable` constraints and the emission of given constraints to bridge the world of dictionaries and the world of kind arrow annotations:

```
mkMatchableInst :: Class -> [Type] -> [PredType] -> TcM ClsInstResult
mkMatchableInst cls [ki, ty] theta = do
  ms <- getAllMatchabilityTyVars ki
  mapM emitUnifiesMatchable ms
  return $ OneInst { cir_new_theta = theta
                   , cir_mk_ev = \_ -> EvExpr (mkCoreConApps (
    classDataCon cls) [Type ki, Type ty])
                   , cir_canonical = EvCanonical
                   , cir_what = BuiltinInstance }
mkMatchableInst _ _ _ = return NoInstance

-- ...

matchUnmatchable :: Class -> [Type] -> TcM ClsInstResult
matchUnmatchable cls [ki, ty] = do
  ms <- getAllMatchabilityTyVars ki
  mapM emitUnifiesUnmatchable ms

  return $ OneInst {
  cir_new_theta = [],
  cir_mk_ev = \_ -> EvExpr (mkCoreConApps (classDataCon cls) [Type (
    typeKind ty), Type ty]),
  cir_canonical = EvCanonical,
  cir_what = BuiltinInstance
}
matchUnmatchable _ _ = return NoInstance
```

Here, `emitUnifiesMatchable` and its counterpart simply emit the given constraint that `m ~ Matchable` or `m ~ Unmatchable` respectively. In the presence of these changes and numerous other mechanical but less relevant changes, the compiler successfully bootstraps, indicating the soundness and backwards compatibility of this approach. However, this success is misleading. Despite being theoretically sound, programs compiled by the bootstrapped compiler demonstrate unusual runtime behaviour.

4.3 Optimisation Attempts

Surprisingly, despite bootstrapping, running programs compiled by the compiler allocate a rather unusual amount of memory. Inspecting the memory allocation profile of these programs indicates rapidly increasing allocations suggestive of naive duplication of evidence terms within the evidence chain.

Naturally, given the trivial nature of `Matchable` and `Unmatchable` dictionaries, it would be sound for GHC to optimise out the dictionaries. Such an optimisation would be reasonable, as trivial dictionaries are a matter of concern only to the type checker and as such, they may be erased alongside other type information. It is perhaps a surprising revelation that GHC does not do this for any boxed constructs that are considered live. The following sections document multiple optimisation strategies at almost every stage of the intermediate representation and show how each stage presents significant architectural constraints.

The desugarer initially seems like a natural place to optimise out trivial boxed data. As the desugarer corresponds to the highest-level intermediate representation, and immediately follows the type checker, it is easy to imagine it is simply a case of generating code as if the Matchable constraints produced by the type checker did not exist. That is, we simply scrub all type signatures of Matchable constraints to erase them.

However, a number of issues stand in the way of successfully performing this optimisation. Firstly, as discussed earlier, the desugarer produces ad-hoc custom Core for certain constructs. The solution here is simple: revert the custom desugaring that introduces new dictionaries and override the type embedded in the various data constructors in the custom core, not unlike the methodology that would be employed in the typical code generation path. However, this is not the only ad-hoc point of code generation of concern.

Core bindings for classes, data constructors and other runtime constructs in Haskell that fall under the umbrella term ”type constructor” are not generated until much later. The bindings are inserted into the main code generation path only after the simplifier stage; that is, two core-to-core passes after the desugarer. Importantly, this means that any dictionary erasure done at the desugarer phase must be duplicated and synchronised with dictionary erasure done at the point of introduction of type constructor bindings.

Given these constraints, we begin with a discussion of introducing such an optimisation pass in the simplifier pass, a intermediate representation within GHC, another potentially natural resting place for such an optimisation.

4.3.1 The Core Simplifier

The simplifier, as the name suggests, already performs a number of lightweight optimisations including β-reduction of lambda terms as a form of partial evaluation, inlining and let-binding floating to potentially expose more optimisation opportunities. The dictionary optimisation pass introduced to the Core simplifier runs before all other Core simplifier passes and acts as a type-directed program transformation that maintains various Core invariants while removing any instance of Matchable binders and evidence terms. This idea is not unlike other dead-code elimination passes that eliminate truly zero-width terms like `proxy#` and `void#`.

However, implementation proves more complex. GHC does not maintain a centralised source of truth for any of the type information within the Core program itself. Despite the presence of a large, central environment aptly named `ModGuts` which contains amongst other things a list of all the type constructors, class instances, pattern synonyms, core bindings and so on, the precise type information is duplicated between the `ModGuts` source and the generated Core bindings also carried around in `ModGuts`.

```
data ModGuts
  = ModGuts {
        {- ... Fields omitted ... -}
        mg_tcs       :: ![TyCon],
        mg_defaults  :: !DefaultEnv,
        mg_insts     :: ![ClsInst],
        mg_fam_insts :: ![FamInst],
        mg_patsyns   :: ![PatSyn],
        mg_rules     :: ![CoreRule],
        mg_binds     :: !CoreProgram,
        mg_foreign   :: !ForeignStubs,
        {- ... Further omission ... -}
        mg_inst_env     :: InstEnv,
        mg_fam_inst_env :: FamInstEnv,
        {- ... Further omission ... -}
    }
```

Every single one of the presented record fields will have a set of cached type information pertaining to the construct embedded within each type constructor, instance, or other construct available in `ModGuts`. Hence, one of two approaches must be taken: either all of the presented record fields will need to be updated in a synchronised manner or type information sourced from any of these record fields that may reach code generation must be intercepted.

The full optimisation pass built is large, so the vast majority of the code is omitted. With that in mind, a simplified form of the algorithm to perform the optimisation pass is shown below:

```
removeMatchablePass :: CoreProgram -> CoreProgram
removeMatchablePass = map transformBind

transformBind :: CoreBind -> CoreBind
transformBind (NonRec b e) =
  NonRec (scrubBinder b) (transformExpr e)
transformBind (Rec pairs) =
  Rec [(scrubBinder b, transformExpr e) | (b, e) <- pairs]

removeMatchableThetas :: Type -> (Type, Int)
removeMatchableThetas ty =
  let (binders, constraints, body, casts) = decompose ty
      constraints' = filter (not . isMatchableTheta) constraints
      removed = length constraints - length constraints'
  in (reconstruct binders constraints' body casts, removed)
```

```
transformExpr :: CoreExpr -> CoreExpr

-- Skip an argument if it is matchable evidence
transformExpr (App f arg)
  | isMatchableDict arg = transformExpr f
  | otherwise = App (transformExpr f) (transformExpr arg)

-- Skip a binder if it is a matchable binder
transformExpr (Lam b e)
  | isMatchableVar b = transformExpr e
  | otherwise = Lam (scrubId b) (transformExpr e)
transformExpr (Case e b ty alts) =
  Case (transformExpr e)
       (scrubId b)
       (fst (removeMatchableThetas ty))
       (map transformAlt alts)
transformExpr e = descend transformExpr e

-- Update data constructors to remove Matchable fields and update the
    arity.
updateDataCon :: DataCon -> DataCon
updateDataCon dc = dc
  {
        {- Omitted for brevity -}
  , dcRep = --- ... ?
  }
```

Of concern to this pass is the reconstruction of any `DataCon`[1] encountered in this optimisation pass. As data constructors also become runtime constructs that take arguments and produce new data, the runtime representation of a data constructor (as denoted `dcRep`) is an important consideration. The field `dcRep` has the type `DataConRep`, which is defined as follows:

```
data DataConRep
  = -- NoDataConRep means that the data con has no wrapper
    NoDataConRep
    -- DCR means that the data con has a wrapper
  | DCR { dcr_wrap_id :: Id
        , dcr_boxer   :: DataConBoxer
        , dcr_arg_tys :: [Scaled Type]
    }
```

Data constructors have both so-called wrappers and workers to construct the data constructor. Wrappers are generally the point-of-call for a data constructor and take arguments that are either boxed or unboxed depending on strictness. The wrapper will then perform whatever relevant unboxing is required to pass arguments to the worker and the worker will perform the relevant computation.

Where pattern matching on a data constructor occurs in code, the data within the data constructor worker representation must be collected and re-boxed where relevant, to be used in

[1]This is the internal representation of a Data Constructor in GHC.

the rest of the program. The part of the data constructor representation responsible for this is `DataConBoxer`, which is defined as follows:

```
-- | Data Constructor Boxer
newtype DataConBoxer = DCB (
    [Type] -> [Var] -> UniqSM ([Var], [CoreBind])
)
```

Surprisingly, the boxer is an opaque function with the intention of being executed with the arguments to the Data Constructor to produce boxing judgements. Instead, a map of boxers per argument would have been more amenable to modification as the number of arguments to the data constructor are statically known during construction[2]. However, with this in mind, the problem now becomes that the function encoded within the `DataConBoxer` now expects an opaque but predetermined number of arguments. This is unfortunately rather difficult to reconcile with the erasure of arguments.

An alternative would be to reconstruct the opaque function on-the-fly. However, the initial construction of the boxer in `buildDataCon`, often performed in the type checking phase, relies on complex logic to determine the exact wrapping and unwrapping boxing judgements for each argument. Refactoring this logic such that it is accessible to both the type checker and the simplifier would potentially require non-trivial architectural changes[3].

Unfortunately, not preserving the exact boxing judgements necessary for the data constructor will corrupt the memory representation. After such corruption, the garbage collector will fail to evacuate data regarding the data constructor due to the fact the memory representation is unrecognisable, failing an internal invariant within the garbage collector.

4.3.2 Unarise

GHC successfully optimises out truly zero-width unboxed types like `proxy#` and `void#`. Given the memory layout challenges encountered in the simplifier, a potential alternative is to treat `Matchable` dictionaries as zero-width, allowing existing optimisation infrastructure in GHC to handle them. Such optimisation happens in an intermediate representation called STG, in an optimisation pass called Unarise. Unarise is responsible for consistently representing all zero-width types as `void#` and removing them where it is safe to do so.

However, this approach does not work for similar reasons. Unarise does not remove zero-width arguments from function arguments or binders to preserve calling conventions with regards to the arity of the function, instead just opting to consistently set the argument to `void#`. With data constructors, however, unarise will remove the zero-width arguments. In the available GHC documentation in the codebase, the latter decision is justified in that "DataCon applications in STG are always saturated.". This explanation documents well the behaviour, but not the underlying rationale.

[2]See (Hinze 2000) for an overview of type-indexed structures.

[3]Alternatively, this limitation may be overcome by pre-composing the boxer with functions such that dummy types (resp. dummy variables) are inserted in the position that a matchable type (resp. variables) were expected, and post-composing such that the bindings are filtered out for deleted arguments. However, given the existence of a functional alternative approach and time constraints, investigating this is left as a matter of future work.

Four approaches were exhaustively attempted: removing binders and arguments in unarise, replacing them with the zero-width `void#`, replacing them with the word-sized unboxed term `nullAddr#` and finally replacing with () to attempt to encourage sharing.

Removing binders and arguments produced a similar issue to the issue that arised in the simplifier pass: garbage collection would get confused due to the unusual memory representation. This is unsurprising, given the data constructor boxer has not been updated. Replacing the dictionaries with the zero-width `void#` was also cause for concern: while garbage collection would now recognise the closure for the data constructor or function, memory alignment in the layout for the closure was off by one byte and therefore swiftly caused memory corruption and a segmentation fault.

Replacing the argument with the word-sized unboxed term `nullAddr#` works to preserve memory alignment. However, likewise, the garbage collector expects an boxed term in this memory area and therefore during evacuation will attempt to free a null pointer. Finally, replacing with `()` causes similar uncontrolled allocation. In this case, it is the runtime system rapidly allocating stack frames, suggesting infinite recursion within the RTS. When limiting the stack allocation budget for the runtime system and attempting to dump stack allocations, the runtime hangs and crashes with an internal assertion failure within the garbage collector. Namely, such a failure happens in `MarkWeak.h`, enforcing the following assertion:

```
ASSERT(get_itbl((StgClosure *)t)->type == TSO);
```

This assertion failure suggests a closure that was expected to be a thread state object has been corrupted, suggesting a rather grave corruption of the RTS itself in face of the swath of dictionaries. This potentially suggests the RTS is not capable of handling such a stress test, despite the lack of heap allocations, and warrants future work for GHC.

In short, none of the above approaches are successful in erasing trivial data without encountering further issues, suggesting perhaps why GHC does not yet admit this optimisation. It is worth noting that should the boxer for the data constructor be represented in a more sensible manner, all of these issues would likely be rather quickly avoided. Unfortunately, however, this design decision has been baked into GHC.

4.3.3 Identifying RTS Sensitivity

It is tempting to attribute the memory allocation explosion to the possibility that GHC, or the RTS, is not aware that it should share the dictionaries. To test this hypothesis, a final attempt at optimising the evidence terms was employed as a `Cmm` pass. In short, a static closure binding present in all compiled executables and allocated in the `.data` section of the executable was employed. Instead of having all dictionaries represent possibly separate heap allocations, a quick pass over the generated `Cmm` code would point all dictionaries to the static closure.

In short, during the building of a data constructor in `Cmm`, the case for a matchable evidence term is special cased:

```
buildDynCon' binder mn actually_bound ccs con args
  | Just cls <- tyConClass_maybe (dataConTyCon con)
  , className cls == matchableClassName
  = do { platform <- getPlatform
       ; let label = mkCmmClosureLabel rtsUnitId (fsLit "stg_MATCHABLE_
    DICT_closure")
       ; return (litIdInfo platform binder (mkConLFInfo con)
                           (CmmLabel label),
                 return mkNop)}
```

Where the literal points to a built-in static closure `stg_MATCHABLE_DICT_closure` built into the executable. Unfortunately, despite this, the RTS still exhibited similar runtime behaviour in allocation, suggesting a latent issue within the RTS itself.

4.4 Unification of Defaulting Rules

Given the various adverse interactions the above solution had with RTS, a different solution is necessary. In Chapter 3, a discussion was presented on the simplification of defaulting rules. Such a simplification relied on the inspection of the structure of a concrete type-level function before assigning the kind a specific marker.

The algorithm for such begins with tooling to set unmatchable annotations along the spine of a type, where a spine is defined as the top level set of kind arrows.

```
setUnmatchable :: Maybe Int -> Type -> Type

setUnmatchable (Just 0) ty = ty

setUnmatchable res_k (ForAllTy bndr ty) =
  ForAllTy bndr (setUnmatchable res_k ty)
setUnmatchable res_k (FunTy af mult mat arg res) =
  case res_k of
    Just n -> FunTy af mult unmatchableDataConTy
                    arg
                    (setUnmatchable (Just (n-1)) res)
    Nothing -> FunTy af mult unmatchableDataConTy
                     arg
                     (setUnmatchable Nothing res)
setUnmatchable res_k (AppTy t1 t2) =
  AppTy (setUnmatchable res_k t1) (setUnmatchable res_k t2)
setUnmatchable res_k (TyConApp tc args) =
  TyConApp tc (map (setUnmatchable res_k) args)
setUnmatchable res_k (CastTy ty co) =
  CastTy (setUnmatchable res_k ty) co
setUnmatchable _ ty = ty
```

Then, specifically for the case of concrete type-level functions, we intercept the construction of the type checker representation for the type constructor and determine which kind arrows to set to unmatchable along the spine:

```
mkTyCon :: Name -> [TyConBinder] -> Kind -> [Role] -> TyConDetails ->
    TyCon
mkTyCon name binders res_kind roles details
  | isMatchableTyCon tc Nominal = tc
  | otherwise = tc { tyConKind = setUnmatchable (Just (spine_full -
    spine_res)) (mkTyConKind binders res_kind) }
  where
    tc = TyCon { tyConName              = name
               , tyConUnique            = nameUnique name
               , tyConBinders           = binders
               , tyConResKind           = res_kind
               , tyConRoles             = roles
               , tyConDetails           = details
               , tyConKind              = mkTyConKind binders res_kind
               , tyConArity             = length binders
               , tyConNullaryTy         = mkNakedTyConTy tc
               , tyConHasClosedResKind = noFreeVarsOfType res_kind
               , tyConTyVars            = binderVars binders }
    spine_res = countSpine res_kind
    spine_full = countSpine (mkTyConKind binders res_kind)
```

With such a mechanism in place, and hence without the difficulties that had arisen in carrying the evidence through a dictionary, all backwards compatibility is retained and a robust solution arises.

Chapter 5

On Halting

5.1 Finite-Base System FO

System FO is an unusual language. The language is decidable[1] and yet a fixed point combinator is present in the syntax and semantics of the language. The distinction between strong normalisation and decidability is key; a strongly normalising language is decidable, but the converse does not hold.

Untyped lambda calculus (ULC) is a canonical example of a Turing-complete and therefore undecidable language. ULC admits an encoding of the higher-order Y-combinator within the language and in the presence of no other restrictions, may encode a Turing machine through various clever encodings. Not unlike ULC, Programmable Computable Functions (PCF) simply admit a built-in constant `Fix` atop a syntax that is essentially an extension of the simply typed lambda calculus (STLC) (Loader 2001; Plotkin 1977). Rather unsurprisingly, System FC, the internal specification for Haskell's Core language, is also Turing-complete with the introduction of recursive let bindings (Weirich, Hsu, and Eisenberg 2013).

The apparent undecidability of models of computation that express day-to-day computation is striking. Where the underlying language of concern is undecidable, type checking a fully dependently-typed variation of the language is also rendered undecidable (Eisenberg 2016). Similarly, optimising compilers that perform dead code elimination or loop unrolling are faced with a difficult choice: compromise the scope of the optimisation or face rendering the termination of the optimiser itself undecidable. Partial evaluation, and supercompilation, are hampered in the presence of undecidability; even pre-evaluating computations that are not dependent on the program input is undecidable and therefore must be left unevaluated.

Often, inadequate solutions are proposed in the form of "fuel" based approaches that allow such evaluation up to a fixed number of steps before bailing (Eisenberg 2016). Unfortunately, such an approach will bail for terminating programs that take at least the predetermined number of steps. The alternative is to restrict recursive programs to primitive recursion or that of which can be statically verified to terminate through a "sized types" based approach (Abel 2010).

[1]In this chapter, and throughout this thesis, "decidable" refers to the decidability of termination of β-reduction. That is, the β-equality of two terms.

Therefore, the introduction of System FO constitutes a novel language capable of maintaining a surprising level of expressivity while nevertheless remaining decidable. To begin describing System FO, we first describe finite-base System FO. Finite-base System FO can be considered related to finitary PCF, which has been proven to be undecidable (Loader 2001), in such way that finite-base System FO recovers the vast majority of the expressiveness of finitary PCF.

5.1.1 Definitions

First, consider the following type syntax:

Definition 5.1. The Type Syntax of Finite-Base System FO

$$\begin{aligned}
\tau_0 &::= B \\
\tau_{k+1} &::= \tau_k \\
&\mid \tau_k \ + \ \tau_k \ (+ \ \tau_k)^N \\
&\mid \tau_k \ \times \ \tau_k \ (\times \ \tau_k)^N \\
&\mid \tau_k \ \rightarrow \ \tau_k \ (\rightarrow \ \tau_k)^N \\
&\mid \forall(a : \tau_k).\ (\forall(b : \tau_k).)^N \ \ \tau_k
\end{aligned}$$

Here, define B to be the syntax the base type. Now consider the set $\mathtt{Type}_0$ to be the set of types represented by the syntax B. As alluded to by the name, we mandate that the set $\mathtt{Type}_0$ has finite cardinality. For all $k \geq 1$, we define τ_k to represent order-k syntax and $\mathtt{Type}_\mathtt{k}$ to define the set of types represented by the syntax τ_k. The superscript N denotes the syntax allowing up to N-many applications of the aforementioned constructors for some fixed parameter N, which parametrises the width of the applications of constructors within the syntax and therefore the final syntax for finite-base System FO. This parametrisation does not constrain practical programs of finite size due to well-typed finite terms having finite width types.

The unusual type stratification is a type stratification of order. We may synonymously call rank-k types order-k types interchangeably and the reason for such will become apparent later.

Definition 5.2. The Term Syntax of Finite-Base System FO

Let the syntax of finite-base System FO be defined as follows:

$$\begin{aligned}
e ::= {} & x \mid \lambda x : \tau.\ e \mid \Lambda a.e \mid e\ e \mid e\ \tau \mid c \mid \mathtt{Fix}\ e \\
& \mid \mathtt{inl}\ e \mid \mathtt{inr}\ e \mid (e, e) \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \\
& \mid \mathtt{case}\ e\ \mathtt{of}\ \{\ (e \mapsto e;)^+\ (\_ \mapsto e;)\ \}
\end{aligned}$$

Two particular pieces of syntax are introduced: a fixed-point combinator `Fix` and a pattern-matching case analysis clause.

Case analysis clauses are mandated to be exhaustive and, at least syntactically, this is enforced through a default clause that is always present in any case analysis. Where the scrutinee does not match any non-default branch, it falls through to the default. The fixed point operator, unsurprisingly, is the construct that enables the semantics of recursive computation.

Definition 5.3. The Typing Rules of Finite-Base System FO

Let the typing rules of finite-base System FO be defined as follows:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{c \in T \quad T \in B}{\Gamma \vdash c : T}$$

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2}{\Gamma \vdash (\lambda x : \sigma_1 .\, e) : (\sigma_1 \to \sigma_2)}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \to \sigma_2 \qquad \Gamma \vdash e_2 : \sigma_1}{\Gamma \vdash e_1 e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \in \tau_k}{\Gamma \vdash \Lambda a .\, e : (\forall a \in \tau_k) .\, \sigma}$$

$$\frac{\Gamma \vdash e : (\sigma \to \sigma) \to \sigma \to \sigma}{\Gamma \vdash \texttt{Fix}\, e : \sigma \to \sigma}$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash p_i : \sigma \quad \Gamma \vdash e_i : \tau \quad \Gamma \vdash e_{\text{default}} : \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{p_1 \mapsto e_1; \ldots; p_n \mapsto e_n; \_ \mapsto e_{\text{default}}\} : \tau}$$

$$\frac{\Gamma \vdash e : \sigma_1}{\Gamma \vdash \text{inl}\, e : \sigma_1 + \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma_2}{\Gamma \vdash \text{inr}\, e : \sigma_1 + \sigma_2}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (e_1, e_2) : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \text{fst}\, e : \sigma_1}$$

$$\frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \text{snd}\, e : \sigma_2}$$

Most of the above typing rules are reminiscent of those of System F. However, the typing rule for `Fix` is quite unusual from the typical presentations of `Fix` in other languages. The distinction is key for the preservation of well-typedness under β-reduction but does not effect the general power of `Fix`; the type for e is a recursive function precisely in continuation passing style. That is, the first argument is a continuation and the second the input to the recursive function.

Definition 5.4. The Equational Theory of Finite-Base System FO

The equational theory of the language is largely that of $\alpha\beta\eta$-equivalence in System F adapted to the stratified type system. Firstly, consider the following definition of α-equivalence:

$$\begin{aligned} \lambda x : \sigma.\, M &=_\alpha \lambda y : \sigma.\, M[x := y] && \text{where } y \notin FV(M) \\ \Lambda a.\, M &=_\alpha \Lambda b. M[a := b] && \text{where } b \notin FV(M) \end{aligned}$$

This definition is typical and is simply the definition of α-equivalence within System F. Now, consider the following equation for β-equivalence. We denote unmatched branches with the $(\ldots)$ within the case branches.

$$\begin{aligned} (\lambda x : \sigma.\, M)\; N &=_\beta M[x := N] \\ (\Lambda a.\, M)\; \tau &=_\beta M[a := \tau] \\ (\texttt{Fix}\, e) &=_\beta e\; (\texttt{Fix}\, e) \\ (\texttt{Fix}\, e) &=_\beta (\texttt{Fix}\, e') \quad \text{if } e =_\beta e' \\ \texttt{fst}\; (e_1, e_2) &=_\beta e_1 \\ \texttt{snd}\; (e_1, e_2) &=_\beta e_2 \\ \texttt{case}\; e\; \texttt{of}\; (\ldots;\; p \mapsto e';\; \ldots) &=_\beta e' \quad \text{if } e =_\beta p \\ \texttt{case}\; e\; \texttt{of}\; (\ldots;\; \_ \mapsto e';) &=_\beta e' \quad \text{if no pattern matches} \end{aligned}$$

Notice the reduction for the fixed-point `Fix e`. The reduction is the usual reduction for a fixed-point combinator and under this reduction rule, the threading of a recursive continuation into e becomes apparent. We also have the following equation for η-equivalence hold under the context $\Gamma \vdash M : \tau_k \rightarrow \tau_j$ and x does not appear free in M.

$$\begin{aligned} (\lambda x : \sigma.\, Mx) &=_\eta M \\ (\Lambda x.\, Mx) &=_\eta M \end{aligned}$$

Definition 5.5. Homeomorphic Embedding of Finite-Base System FO Terms with Finite Labelling

Consider Kruskal's tree theorem, which states that any infinite trace of finitely-labelled trees is guaranteed to have a pair of embedding trees. We define the function $\texttt{tree} : \Lambda \rightarrow \texttt{Tree}_\Lambda$ as follows, where Λ is the set of all finite-base System FO terms and $\texttt{Tree}_\Lambda$ is the set of all finite tree constructions of System FO terms:

$$\begin{aligned}
\texttt{tree}(x) &= \texttt{Label}(x) \\
\texttt{tree}(c) &= \texttt{Label}(c) \\
\texttt{tree}(\texttt{Fix}\, e) &= \texttt{Label}(\texttt{enc}(\texttt{Fix}\, e)) \\
\texttt{tree}(\lambda x.M) &= \texttt{Node}_{\lambda}(\texttt{tree}(M)) \\
\texttt{tree}(\Lambda a.M) &= \texttt{Node}_{\Lambda}(\texttt{tree}(M)) \\
\texttt{tree}(e_1 e_2) &= \texttt{Node}_{\mathrm{App}}(\texttt{tree}(e_1),\texttt{tree}(e_2)) \\
\texttt{tree}(\texttt{inl}\; e) &= \texttt{Node}_{\mathrm{inl}}(\texttt{tree}(e)) \\
\texttt{tree}(\texttt{inr}\; e) &= \texttt{Node}_{\mathrm{inr}}(\texttt{tree}(e)) \\
\texttt{tree}((e_1,e_2)) &= \texttt{Node}_{\mathrm{pair}}(\texttt{tree}(e_1),\texttt{tree}(e_2)) \\
\texttt{tree}(\texttt{fst}\; e) &= \texttt{Node}_{\mathrm{fst}}(\texttt{tree}(e)) \\
\texttt{tree}(\texttt{snd}\; e) &= \texttt{Node}_{\mathrm{snd}}(\texttt{tree}(e)) \\
\texttt{tree}(\texttt{case}\; e\; \texttt{of}\; \{p_i \mapsto e_i; \_ \mapsto e_d\}) &= \texttt{Node}_{\mathrm{case}_d}(\texttt{tree}(e),\texttt{tree}(p_1),\texttt{tree}(e_1),\ldots,\texttt{tree}(e_d))
\end{aligned}$$

There is an unusual leaf node in this construction; fix point expressions are also considered leaf nodes. Given a fix point combinator is not constructible as a lambda term in System F, and is only a syntactically defined construct in finite-base System FO, the number of unique `Fix` labels is constrained by the reduction of the e under the `Fix`. We define the set of `Fix`-labels to be the set of unique encodings of `Fix` under the encoding `enc`.

Supposing that the set of leaf labels are finite, Kruskal's tree theorem defines homeomorphic embedding over the above tree construction and proves it is a well-quasi order (WQO). We henceforth use the relation $(\leq_H)$ to denote homeomorphic embedding.

Definition 5.6. Generated Equivalence Classes of Homeomorphic Embedding

Consider the equality relation $(\sim_H)$ defined such that $e_1 \sim_H e_2$ if and only if $e_1 \leq_H e_2$ or $e_2 \leq e_1$. Let the set of equivalence classes generated by $\sim_H$ be called H. The set H has finite cardinality. This is a claim we will prove as Lemma 1.6.

5.1.2 Lemmas & Theorems

The above definitions describe the specification for a finite-base System FO. A key obligation resides with the type system: a proof of decidability.

Lemma 5.7. *(Finite Type Inhabitants): A set of types $Type_k$, for some fixed k, has a finite number of inhabitants.*

Proof. By Induction.

Base Case: Consider the base case for $\mathtt{Type}_0$. By definition, this set of types has a finite number of inhabitants.

Inductive Case
Now, assume the inductive hypothesis for some set of types $\mathtt{Type}_{k-1}$. Consider that some type belonging to $\mathtt{Type}_k$ may be constructed through four cases.

Case τ_{k-1}: Consider the case that some type is constructed by simply choosing a type from $\texttt{Type}_{k-1}$. By the inductive hypothesis, there are finitely many of such types, as required.

Case $\tau_{k-1} + \tau_{k-1}$: Consider the case of types generated through the sum type constructor $(+)$. Suppose there are K applications of such a constructor. Let N_{k-1} denote the cardinality of $\texttt{Type}_{k-1}$. There are KN_{k-1} unique types constructible by the sum type constructor $(+)$ of the form $\tau_{k-1} + \tau_{k-1}$, and thus there are finitely many of these types.

Case $\tau_{k-1} \times \tau_{k-1}$: Consider the case of types generated through the product type constructor $(\times)$. Let N_{k-1} denote the cardinality of $\texttt{Type}_{k-1}$. Suppose there are K applications of such a constructor. There are N_{k-1}^{K} unique types constructible by the product type constructor $(\times)$ of the form $\tau_{k-1} \times \tau_{k-1}$, and thus there are finitely many of these types.

Case $\tau_{k-1} \rightarrow \tau_{k-1}$: Consider the case of types generated through the function type constructor $(\times)$. Suppose there are K applications of such a constructor. Let N_{k-1} denote the cardinality of $\texttt{Type}_{k-1}$. There are $N_{k-1} \uparrow\uparrow K$ unique types constructible by the function type constructor $(\rightarrow)$ of the form $\tau_{k-1} \rightarrow \tau_{k-1}$, and thus there are finitely many of these types.

Case $\forall(\alpha : \tau_{k-1}).\tau_{k-1}$: Consider the case of types generated through the polymorphism. Suppose there are K applications of such a constructor. Let N_{k-1} denote the cardinality of $\texttt{Type}_{k-1}$. There are $N_{k-1} \uparrow\uparrow K$ unique types constructible, arising from the substitution-like nature of polymorphism, and thus finitely many of these types.

Given the set of types $\texttt{Type}_k$ is simply the union of the above four cases, we show that the entire set likewise has finite cardinality. □

Lemma 5.8. *(Finite Binder Depth): The binder depth of a well-typed term e is finitely bounded.*

Proof. By induction.

Consider the base case of a well-typed term $e : \sigma$ such that $\sigma : \tau_0$. By construction of τ_0, e may not be nor contain a lambda abstraction. Therefore, the binder depth is finitely bounded.

Assume the inductive hypothesis. Consider a well-typed term $e : \sigma$ such that $\sigma : \tau_k$ and enumerate the cases for which the type σ is constructed.

Case τ_{k-1}: Consider the case that some type is constructed by simply choosing a type from $\texttt{Type}_{k-1}$. By the inductive hypothesis, the binder depth is finitely bounded

Case $\tau_{k-1} + \tau_{k-1}$: Consider the case of types generated through the sum type constructor $(+)$. A sum type constructor does not allow a new binder and therefore by the inductive hypothesis over the components, the binder depth is finitely bounded.

Case $\tau_{k-1} \times \tau_{k-1}$: Consider the case of types generated through the product type constructor $(\times)$. A product type constructor does not allow a new binder and therefore by the inductive hypothesis over the components, the binder depth is finitely bounded.

Case $\tau_{k-1} \rightarrow \tau_{k-1}$: Consider the case of types generated through the function type constructor $(\times)$. Suppose there are K applications of such a constructor. Therefore, by the typing rules, there are K new binders. By the inductive hypothesis, the body of the lambda abstraction has a finitely bounded binder depth and thus the term has a whole has a finitely bounded binder depth.

Case $\forall(\alpha : \tau_{k-1}).\tau_{k-1}$: Consider the case of types generated through the polymorphism. Suppose there are K applications of such a constructor. Therefore, by the typing rules, there are K new binders. By the inductive hypothesis, the body of the lambda abstraction has a finitely bounded binder depth and thus the term has a whole has a finitely bounded binder depth.

□

Lemma 5.9. *(Finite Renaming Under α-Substitution): The number of unique binders names of a well-typed term* `e` *is finitely bounded.*

Proof. By Lemma 5.8, the binder depth of a well-typed term e is finitely bounded. Suppose De Bruijn renaming is used as a canonical naming for binders in the term e. By construction, there is a one-to-one correspondence between binders and binder depths. □

Lemma 5.10. *(Finite* `Fix`*-Label Encodings): The number of unique* `Fix`*-label encodings are finitely bounded.*

Proof. By mutual induction on Lemma 5.13.

Base Case: First, consider the base case that there are at most 1-deep nested fix point operators within the term t. Suppose that t is evaluated under an arbitrary evaluation order. Then, the trace $(t, t_1, t_2, \dots)$ must have a sub-trace e' that is the trace for reducing e, where e is the expression under the deepest fix point operator (if there are many, pick an arbitrary one). The term e, by definition, does not have a fix point operator embedded within it. Recall the non-`Fix` fragment of System FO is a stratified subsystem of System. Therefore, by the strong normalisation of System F, the sub-trace e' must be finite. Therefore, there are finitely many expressions that can appear under the `Fix` and therefore finitely many `Fix`-labels.

Inductive Case
Now assume the inductive hypothesis for k-deep nested fix point operators and suppose the inductive case for $(k+1)$-deep nested fix point operators. Suppose that t is evaluated under an arbitrary evaluation order. Then, the trace $(t, t_1, t_2, \dots)$ must have a sub-trace e' that is the trace for reducing e, where e is the expression under the deepest fix point operator (again, if there are many, pick an arbitrary one). The trace for e' is possibly infinite, but by Lemma 5.13 (and mutual induction), the trace for e' if infinite must have a self-embedding pair under homeomorphic embedding. Consider the equality relation $\sim_H$. By Lemma 5.12, there are finitely many generated equivalence classes as denoted by the set H.

Therefore, for each equivalence class, assign a unique index. The set of unique indices is finite. Each member of the equivalence class assumes as it's `Fix`-label the unique index for it's equivalence class.

Therefore, there are finitely many `Fix`-labels, as required. □

Lemma 5.11. *(Finite Leaf Labels): The number of unique leaf labels for some tree $t \in \texttt{Tree}_\Lambda$ is finitely bounded.*

Proof. By Lemma 5.9, the number of unique binder names is finitely bounded. By Lemma 5.10, the number of unique Fix-labels is finitely bounded. By the finiteness of B, there are finitely many term constants. Therefore, the number of unique leaf labels for some tree $t \in \texttt{Tree}_\Lambda$ is finitely bounded. □

Lemma 5.12. *(Finite Equivalence Classes Under $\sim_H$): The number of equivalence classes generated by $\sim_H$ are finitely bounded.*

Proof. By contradiction.

Consider the equality relation $\sim_H$ and the set of equivalence classes H. Suppose for contradiction that there were infinitely many such equivalence classes. Then, one could pick a representative for each equivalence class.

We have $H_i \neq_H H_j$ for all $H_i, H_J \in H$. Therefore, we may construct a sequence of finite trees $H_1, H_2, \ldots$ such that there is no pair for which $H_i =_H H_j$ and as such there no pair for which $H_i \leq H_j$ under homeomorphic embedding, which is a contradiction of Kruskal's tree theorem. □

Lemma 5.13. *(Homeomorphic Embedding of Trace States): Consider a trace of a System FO program $t = (t_1, t_2, \ldots)$. Let $t_{\texttt{tree}} = (\texttt{tree}(\texttt{t}_1), \texttt{tree}(\texttt{t}_2), \ldots)$ denote the sequence of trees that correspond to t. Assuming that there are finitely many* `Fix`*-labels, any infinite trace t must have a pair of terms t_i and t_j, where $i < j$, such that $\texttt{tree}(\texttt{t}_\texttt{i})$ embeds $\texttt{tree}(\texttt{t}_\texttt{j})$ under homeomorphic embedding.*

Proof. The label set for the set of finite trees $\texttt{Tree}_\Lambda$ is finite. Consider any infinite sequence of finite trees $\pi_1, \pi_2, \ldots$, where we have $\pi_i \in \texttt{Tree}_\Lambda$. Consider the possibly finite subsequence $\theta_1, \theta_2, \ldots$ such that $\theta_i \in \texttt{Tree}_\Lambda$ and θ_i is not a label. Suppose this subsequence is finite. Therefore, there must be an infinite subsequence of labels. There are finitely many unique labels and therefore, by pigeonhole, there must be a pair π_m and π_n, where $n > m$, such that $\pi_n = \pi_m$.

Suppose this subsequence is infinite. Therefore, by Kruskal's tree theorem, there must be a pair π_m and π_n, where $n > m$, such that $\pi_n \leq_H \pi_m$.

Then, generate the sequence of trees by setting $\pi_i := \texttt{tree}(t_i)$. □

Theorem 5.14. *(Decidability of Finite Base System FO): The normalisation of a term e under deterministic β-reduction is decidable.*

Proof. First, recall a deterministic reduction strategy is one where the reduction relation $\to_S$ is a partial function, i.e., for each term e there is at most one term e' such that $e \to_S e'$.

Let e be an arbitrary term. By Lemma 5.13, the trace of the reduction of e must infinitely often contain an embedding pair of trees under β-reduction or two identical fix labels.

In the latter case, by determinism of β-reduction, the computation is diverging. Thus, consider the former case.

In the case of one-step deterministic reduction strategies, let $p_i = c_1 \to c_2 \to \cdots \to c_n$ denote a path through the tree to reach a specific constructor or leaf node, where $c_i \in \{L, R\}$ denotes the choice to take the left or right sub-branch. By Lemma 1.2, the length of p_i is always finite. In the case of many-step deterministic strategies, we consider p_i to simply be the subtree of the term that describes a set of paths through the tree.

Therefore, by Kruskal's tree theorem and Lemma 5.8, a trace of paths $p = (p_1, p_2, \dots)$ must have an embedding pair infinitely often. Therefore, an embedding pair in both the embedding of term trees and simultaneously the embedding of paths must occur infinitely often, where simultaneous embedding is defined as the index of the embedding pairs being equal.

Suppose p_i denotes the next redex(s) the arbitrary deterministic reduction strategy employed will reduce. Where simultaneous embedding occurs (i.e., $\texttt{tree}(t_i) \leq_H \texttt{tree}(t_j)$ and $p_i \leq_H p_j$), the determinism of the reduction strategy demands that the same redex will be reduced in the same way, leading to a repeating sequence and thus divergence.

□

5.2 System FO

The previous section describes finite-base System FO and proves the decidability of the language. However, it proposes a rather limiting restriction: the set of base types is finite. Despite the fact that, practically, computation is often performed on limited memory devices, such a restriction is too restrictive to write expressive programs. Instead, this section demonstrates the introduction of countably infinitely inhabited base types where a suitable well-quasi order (WQO) exists on the inhabitants.

5.2.1 Definitions

Definition 5.15. The Type Syntax of System FO

Let D_i be the ith inductive data type defined through finitely-branching constructors and a finite set of data type specific base types. Let B_{D_i} denote the syntax of base types of D_i. Let also τ_{k,D_i} denote the rank-k type syntax for the inductive data type D_i. Consider the following syntax, where $F_1, F_2, \dots, F_n$ are the finitely many finite-branching constructors of the data type.

$$\begin{aligned}
\tau_{0,D_i} &::= D_i \\
\tau_{k+1,D_i} &::= \tau_{k,D_i} \\
&\mid F_1\ \tau_{k,D_i}\ \tau_{k,D_i}\ \cdots\ \tau_{k,D_i} \\
&\mid F_2\ \tau_{k,D_i}\ \tau_{k,D_i}\ \cdots\ \tau_{k,D_i} \\
&\mid \dots \\
&\mid F_n\ \tau_{k,D_i}\ \tau_{k,D_i}\ \cdots\ \tau_{k,D_i}
\end{aligned}$$

Let $\tau_{D_i} = \bigcup_k \tau_{k,D_i}$ denote the total syntax over all k for an inductive data type D_i. Then, we define the type syntax for the calculus terms as usual:

$$\begin{aligned}
\tau_0 &::= B \cup \bigcup_i \tau_{D_i} \\
\tau_{k+1} &::= \tau_k \\
&\mid \tau_k + \tau_k\,(+\,\tau_k)^N \\
&\mid \tau_k \times \tau_k\,(\times\,\tau_k)^N \\
&\mid \tau_k \rightarrow \tau_k\,(\rightarrow\,\tau_k)^N \\
&\mid \forall(a:\tau_k).\,(\forall(b:\tau_k).)^N\ \tau_k
\end{aligned}$$

The inhabitants of $\texttt{Type}_\texttt{k}$ are infinite for $k \geq 0$ which, quite unfortunately, means a large number of results regarding finite-base System FO are no longer applicable.

Definition 5.16. The Term Syntax of System FO

Let the term syntax of System FO be defined as precisely that of finite-base System FO.

Definition 5.17. The Typing Rules of System FO

Let the typing rules of System FO be defined as precisely that of finite-base system FO.

Definition 5.18. The Equational Theory of System FO

The equational theory of System FO is that of finite-base System FO.

Definition 5.19. Homeomorphic Embedding of System FO Terms

Kruskal's tree theorem, in the infinite label case, states that any infinite trace of labelled trees is guaranteed to have a pair of embedding trees should the labels form a WQO. In our construction, we mandate that the set defined by the greatest fixed point of the functor of the inductive data type must have a WQO defined on it. We define the function $\texttt{tree} : \Lambda \rightarrow \texttt{Tree}_\Lambda$ as follows, where Λ is the set of all finite-base System FO terms and $\texttt{Tree}_\Lambda$ is the set of all finite tree constructions of System FO terms:

$$\begin{aligned}
\texttt{tree}(x) &= \texttt{Label}(x) \\
\texttt{tree}(c) &= \texttt{Label}(c) \\
\texttt{tree}(\texttt{Fix}\,e) &= \texttt{Label}(\texttt{enc}(\texttt{Fix}\,e)) \\
\texttt{tree}(\lambda x.M) &= \texttt{Node}_\lambda(\texttt{tree}(M)) \\
\texttt{tree}(\Lambda a.M) &= \texttt{Node}_\Lambda(\texttt{tree}(M)) \\
\texttt{tree}(e_1 e_2) &= \texttt{Node}_{\text{App}}(\texttt{tree}(e_1),\texttt{tree}(e_2)) \\
\texttt{tree}(\texttt{inl}\ e) &= \texttt{Node}_{\text{inl}}(\texttt{tree}(e)) \\
\texttt{tree}(\texttt{inr}\ e) &= \texttt{Node}_{\text{inr}}(\texttt{tree}(e)) \\
\texttt{tree}((e_1,e_2)) &= \texttt{Node}_{\text{pair}}(\texttt{tree}(e_1),\texttt{tree}(e_2)) \\
\texttt{tree}(\texttt{fst}\ e) &= \texttt{Node}_{\text{fst}}(\texttt{tree}(e)) \\
\texttt{tree}(\texttt{snd}\ e) &= \texttt{Node}_{\text{snd}}(\texttt{tree}(e)) \\
\texttt{tree}(\texttt{case}\ e\ \texttt{of}\ \{p_i \mapsto e_i;\_ \mapsto e_d\}) &= \texttt{Node}_{\text{case}_d}(\texttt{tree}(e),\texttt{tree}(p_1),\texttt{tree}(e_1),\ldots,\texttt{tree}(e_d))
\end{aligned}$$

We will prove that such a tree structure forms a WQO under homeomorphic embedding.

Definition 5.20. Generated Equivalence Classes of Homeomorphic Embedding

Consider the equality relation $(\sim_H)$ defined such that $e_1 \sim_H e_2$ if and only if $e_1 \leq_H e_2$ or $e_2 \leq e_1$. Let the set of equivalence classes generated by $\sim_H$ be called H. The set H has finite cardinality, as shown in S5.1.

Definition 5.21. Tree Shapes

We define a tree shape of a term t to be the tree $\texttt{tree}(t)$ with all labels set to $\bot$. Concretely, define the function shape as follows:

$$\begin{aligned}
\texttt{shape}(x) &= \bot \\
\texttt{shape}(c) &= \bot \\
\texttt{shape}(\texttt{Fix}\, e) &= \bot \\
\texttt{shape}(\lambda x.M) &= \texttt{Node}_\lambda(\texttt{shape}(M)) \\
\texttt{shape}(\Lambda a.M) &= \texttt{Node}_\Lambda(\texttt{shape}(M)) \\
\texttt{shape}(e_1 e_2) &= \texttt{Node}_{\text{App}}(\texttt{shape}(e_1), \texttt{shape}(e_2)) \\
\texttt{shape}(\texttt{inl}\, e) &= \texttt{Node}_{\text{inl}}(\texttt{shape}(e)) \\
\texttt{shape}(\texttt{inr}\, e) &= \texttt{Node}_{\text{inr}}(\texttt{shape}(e)) \\
\texttt{shape}((e_1, e_2)) &= \texttt{Node}_{\text{pair}}(\texttt{shape}(e_1), \texttt{shape}(e_2)) \\
\texttt{shape}(\texttt{fst}\, e) &= \texttt{Node}_{\text{fst}}(\texttt{shape}(e)) \\
\texttt{shape}(\texttt{snd}\, e) &= \texttt{Node}_{\text{snd}}(\texttt{shape}(e)) \\
\texttt{shape}(\texttt{case}\, e\, \texttt{of}\, \{p_i \mapsto e_i; \_ \mapsto e_d\}) &= \texttt{Node}_{\text{case}_d}(\texttt{shape}(e), \texttt{shape}(p_1), \texttt{shape}(e_1), \ldots, \texttt{shape}(e_d))
\end{aligned}$$

Definition 5.22. Reduction Paths

In the case of one-step deterministic reduction strategies, let $p_i = c_1 \to c_2 \to \cdots \to c_n$ denote a path through the tree to reach a specific constructor or leaf node, where $c_i \in \{L, R\}$ denotes the choice to take the left or right sub-branch, and the path p_i denotes the next redex to be reduced by the reduction strategy. In the case of many-step deterministic strategies, we consider p_i to simply be the subtree of the term that describes a set of paths through the tree that denote the next redexes to be reduced simultaneously by the reduction strategy.

5.2.2 Lemmas & Theorems

Lemma 5.23. *(Finite Binder Depth): The binder depth of a well-typed term* e *is finitely bounded.*

Proof. By induction.

Consider the base case of a well-typed term $e : \sigma$ such that $\sigma : \tau_0$. By construction of τ_0, e may not be nor contain a lambda abstraction. Therefore, the binder depth is finitely bounded.

Assume the inductive hypothesis. Consider a well-typed term $e : \sigma$ such that $\sigma : \tau_k$ and enumerate the cases for which the type σ is constructed.

Case τ_{k-1}: Consider the case that some type is constructed by simply choosing a type from Type$_{k-1}$. By the inductive hypothesis, the binder depth is finitely bounded

Case $\tau_{k-1} + \tau_{k-1}$: Consider the case of types generated through the sum type constructor $(+)$. A sum type constructor does not allow a new binder and therefore by the inductive hypothesis over the components, the binder depth is finitely bounded.

Case $\tau_{k-1} \times \tau_{k-1}$: Consider the case of types generated through the product type constructor $(\times)$. A product type constructor does not allow a new binder and therefore by the inductive hypothesis over the components, the binder depth is finitely bounded.

Case $\tau_{k-1} \rightarrow \tau_{k-1}$: Consider the case of types generated through the function type constructor $(\times)$. Suppose there are K applications of such a constructor. Therefore, by the typing rules, there are K new binders. By the inductive hypothesis, the body of the lambda abstraction has a finitely bounded binder depth and thus the term has a whole has a finitely bounded binder depth.

Case $\forall(\alpha : \tau_{k-1}).\tau_{k-1}$: Consider the case of types generated through the polymorphism. Suppose there are K applications of such a constructor. Therefore, by the typing rules, there are K new binders. By the inductive hypothesis, the body of the lambda abstraction has a finitely bounded binder depth and thus the term has a whole has a finitely bounded binder depth. □

Lemma 5.24. *(Finite Renaming Under α-Substitution): The number of unique binders names of a well-typed term e is finitely bounded.*

Proof. By Lemma 5.23, the binder depth of a well-typed term e is finitely bounded. Suppose De Bruijn renaming is used as a canonical naming for binders in the term e. By construction, there is a one-to-one correspondence between binders and binder depths. □

Lemma 5.25. *(Finite* `Fix`*-Label Encodings): The number of unique* `Fix`*-label encodings are finitely bounded.*

Proof. By the same reasoning as finite base System FO. □

Lemma 5.26. *The number of unique tree shapes in the trace of a program is finite*

Proof. By induction.

Consider the base case for some program that is constructed from terms $e : \sigma$ such that $\sigma : \tau_0$. Then, by construction, the program has no redexes and therefore has a finite number of unique shapes.

Assume the inductive hypothesis. Consider the case for some programs constructed from terms $e : \sigma$ such that $\sigma : \tau_k$. All redexes in the program of the form $(\lambda x.M)N$ will have $M : \sigma_1$ such that $\sigma_1 : \tau_j$ where $j < k$ and $N : \sigma_2$ such that $\sigma_2 : \tau_i$ such that $i < k$. Therefore, the term $M[x := N]$ will be constructed of terms $e' : \sigma'$ such that $\sigma' : \tau_n$ such that $n < k$. By the inductive hypothesis, all terms $M[x := N]$ have finitely bounded shapes.

All redexes in the program of the form or $(\Lambda a.M)\ t$ will have will have $M : \sigma_1$ such that $\sigma_1 : \tau_j$ where $j < k$ and $t : \tau_k$. Therefore, the term $M[a := t]$ will be constructed of terms $e' : \sigma'$ such that $\sigma' : \tau_n$ such that $n < k$. By the inductive hypothesis, all terms $M[x := N]$ have finitely bounded shapes.

All redexes in the form of `Fix` e are represented by a label and therefore have finitely unique shapes by Lemma 5.25.

All redexes of the form `fst` (e_1, e_2) will reduce to $e_1 : \sigma$ such that $\sigma : \tau_f$ and $f < k$. Hence, follows from IH. All redexes of the form `snd` (e_1, e_2) will reduce to $e_2 : \sigma$ such that $\sigma : \tau_s$ and $s < k$. Hence, follows from IH.

All redexes of the form `case` e `of` $(\ldots;\ \_ \mapsto e';) =_\beta e'$ or `case` e `of` $(\ldots;\ p \mapsto e';\ \ldots) =_\beta e'$ will reduce to $e' : \sigma$ such that $\sigma : \tau_c$ and $c \leq k$. There are only a finite number of nested cases in a finite program and therefore after finite reductions, the term will reduce to some e'' that is not a case term. By the previous cases, the term e'' has a finite number of shapes. □

Lemma 5.27. *(Trees form a WQO): The trees corresponding to a program state in a trace form a WQO*

Proof. By Lemma 5.26, there are a finite number of unique shapes programs may take during reduction, and therefore a finite number of $\perp$ terms to represent substitutions with inductive data types.

Let each shape be represented as a unique finitely branching constructor taking as input each inductive data term. As there are finitely many finitely branching constructors, and the signature of base types form a WQO by structural ordering on constructors, the finite trees containing these labels form a WQO under homeomorphic embedding. □

Lemma 5.28. *(Case Default Embedding) Suppose a trace contains a pair of states t_1 and t_2 such that $t_1 \leq_H t_2$ and a path embedding $p_1 \leq_H p_2$ such that if the reduction paths point to a case redex, then the redex will match the default branch, or the trace contains a pair of states such that $t_1 = t_2$. Therefore this trace will reduce infinitely long.*

Proof. Assume the precedent. Let $<_H$ denote non-reflexive homeomorphic embedding, such that $a <_H b$ if and only if $a \leq_H b$ and $a \neq b$. Therefore, either the states are such that $t_1 = t_2$, in which case the trace will repeat due to the determinism of β-reduction, or the states are such that $t_1 <_H t_2$ in which case due to the monotonicity of the default branch and the determinism of β-reduction, the trace will repeat. In both cases, the trace will reduce infinitely long. □

Lemma 5.29. *(Non-Case Default Embedding) Suppose a trace contains a pair of states t_1 and t_2 such that $t_1 \leq_H t_2$ and a path embedding $p_1 \leq_H p_2$ such that if the reduction paths point to a case redex, then the redex will not match the default branch, or the trace contains a pair of states such that $t_1 = t_2$. Therefore this trace will either terminate, or there will eventually be a pair $t_3 \leq_H t_4$ and a path embedding $p_3 \leq_H p_4$ such that if the reduction paths point to a case redex, then the redex will match the default branch, or there will eventually be a pair such that $t_3 = t_4$.*

Proof. Assume the precedent. Let $<_H$ denote non-reflexive homeomorphic embedding, such that $a <_H b$ if and only if $a \leq_H b$ and $a \neq b$. Therefore, either the states are such that $t_1 = t_2$ in which case the trace will repeat due to the determinism of β-reduction, or the states are such that $t_1 <_H t_2$. In the latter case, if the reduction paths do not point to a case redex, then by determinism of β-reduction and monotonicity of substitution, the trace will repeat. If the reduction paths point to a case redex, then the reduction either terminates or does not. The former trivially proves the lemma, so suppose the latter. If the trace does not terminate, then there will either eventually be a pair $t_3 \leq_H t_4$ and a path embedding $p_3 \leq_H p_4$ such that if the reduction paths point to a case redex, then the redex will match the default branch or there will not. The former trivially proves the lemma, so suppose the latter.

Under these assumptions, the trace does not terminate and there will never eventually be a pair $t_3 \leq_H t_4$ and a path embedding $p_3 \leq_H p_4$ such that if the reduction paths point to a case redex, then the redex will match the default branch. Then, all scrutinees of the inductive data type sort for case statements must never match the default branch. There are finitely many non-default branches and by assumption, the trace is infinite, therefore demanding that there must be a pair of states such that $t_3 = t_4$ by pigeonhole. □

Lemma 5.30. *(Soundness of a Whistle): Suppose W is a whistle that defines all traces that do not contain a simultaneous embedding pair of states $t_i \leq_H t_j$ and reduction paths $p_i \leq_H p_j$ such that $i < j$, nor contain a pair of states such that $t_i = t_j$ where $i < j$. Then, if a trace $t \notin W$, then the trace will diverge.*

Proof. By Lemma 5.28 and Lemma 5.29. □

Lemma 5.31. *(Completeness of a Whistle): Suppose W is a whistle that defines all traces that do not contain a simultaneous embedding pair of states $t_i \leq_H t_j$ and reduction paths $p_i \leq_H p_j$ such that $i < j$ such that if p_i and p_j point to a case redex, it is either the case that the redex will match the default branch or $t_i = t_j$ and $p_i = p_j$, nor contain a pair of states such that $t_i = t_j$ where $i < j$. If a trace t diverges, then $t \notin W$.*

Proof. By Kruskal's tree theorem, any infinite trace t will contain a pair of states $t_i \leq_H t_j$ and reduction paths $p_i \leq_H p_j$ such that $i < j$. Where p_i and p_j do not point to case redexes, the lemma is proven. Suppose therefore that they do. Where p_i and p_j point to case redexes, they either match the default branch or they do not. In the former case, the lemma is proven. Suppose therefore they do not. By Lemma 5.29, there will eventually be a pair such that $t_i = t_j$ and $p_i = p_j$, as required. □

Theorem 5.32. *(Decidability of System FO): System FO is decidable*

Proof. By Lemma 5.30 and Lemma 5.31 □

Chapter 6

Evaluation & Future Work

In this thesis, a number of contributions have been presented. Firstly, two implementation approaches for unsaturated type families have been explored, to which both retain sound backwards compatibility and the latter runs without adverse interactions with the Haskell runtime system. Secondly, a formalisation for System FO is presented, a novel decidable language which retains considerable expressivity.

6.1 Unsaturated Type Families

Both approaches for unsaturated type families prove robust with respect to bootstrapping the compiler. The former approach revealed potential adverse behaviour within the Haskell runtime system that warrants further investigation as well as the need for better support for memory optimisations within the architecture that underlies GHC.

The latter has been proven backwards compatible and shows no unexpected regression within the present GHC test suite. Within the GHC test suite, at the time of writing, there are 10,491 tests. Of these, 127 tests are inherently expected to fail on the commit forked from the main branch. 152 further failures within the test suite in the implementation of the prototype for the support of unsaturated type families are benign, expected failures. These failures are predominantly due to the output of the test case changing slightly, naming conflicts, tests that directly assume unsaturated type families are not supported or unrelated ARM64 linker errors (when tested on the author's machine). Therefore, this fork is on track to being merged into mainstream GHC subject to a formal proposal, review process, extensive testing and enhancing the ergonomics of the implemented features. Indeed, one potential avenue of future work is the adaptation of existing libraries to utilise unsaturated type families to remove now unnecessary boilerplate.

Both approaches explored are well-specified with respect to the System FC formalisation and, after suitable investigation of the latent sensitivities of the Haskell runtime system, are both suitable approaches for the implementation of unsaturated type families. Where the former approach may benefit from certain novel optimisations, this thesis has described the the implementation to enable optimisation pass within the GHC pipeline, and both leaves as future work an investigation into the modification of data constructor memory representations, and suggests an way forward as the final missing piece.

In comparison to the previous two attempts at implementing unsaturated type families, the concerns regarding adverse interaction with Template Haskell (Kiss 2018) and the numerous defaulting rules (Chen 2024) are entirely resolved. The approach proposed finds a universally applicable way of determining the matchability of type-level functions according to the structural and semantic information available to the type checker. In other words, unsaturated type families are finally ready to become a Haskell language extension.

6.2 System FO

System FO is a novel, decidable typed lambda calculus which retains dramatic expressivity through a subtle restriction. This thesis has proposed a method of termination analysis in System FO, online verification of termination, which renders the language decidable.

The implications for this are significant. To the knowledge of this thesis, no typed lambda calculus in current literature admits a fix point combinator in a decidable language. The flexibility of such a language renders the vast majority of real-world code potentially decidable. Where code remains undecidable, defunctionalisation remains on-hand as an apt mitigation for order-stratification violation.

Literature regarding Dependent Haskell has so far accepted soundness issues due to the assumption that type checking Dependent Haskell must be undecidable (Eisenberg 2016). Instead, this thesis proposes a lightly restricted target for Dependent Haskell computation that renders type checking decidable and therefore sound.

Likewise, a large number of compiler optimisations are enabled due to the decidability of the underlying language. As termination of a program is certain to be decidable, compilers may take more aggressive decisions with dead code elimination, constant folding and loop unfolding which have so far been hampered by the assumption of the undecidability of the underlying code. Furthermore, a long standing question in computability is answered: computation can be decidable while recovering significant expressivity.

A number of other avenues of future work are opened up with regards to System FO. Firstly, the informal proof detailed in Chapter 5 is due for mechanisation to verify robustness. Moreover, the practical considerations of implementing a whistle as described must be considered; this thesis conjectures the pathological worst-case time-complexity of decidability of a System FO program is TOWER-complete. Nonetheless, the nature of the pathological worst-case and the actual practical likelihood of such a worst-case occurring are key avenues of further research.

The boundary of decidability that is revealed in the formalisation of Chapter 5 also suggests that future work is warranted in closer documenting the effect that the cardinality of the inhabitants of the base types has on the decidability of a language. Specifically, Chapter 5 proposes the insight that both finite inhabitants of base types may be admitted and retain decidability, and likewise countably infinite inhabitants may be admitted under the pretense they admit a WQO. This suggests a connection between the cardinality of base types and decidable computation, and raises the question of where the precise boundary of decidability holds; this thesis conjectures that any language where the type inhabitants become uncountably infinite will lose decidability.

As Chapter 5 focuses on inductive termination analysis, another further avenue of research is to investigate the implications for coinductive productivity analysis as a dual to this work. Likewise, investigating the existence of safe decidability boundaries within otherwise Turing-complete languages would allow practical users of a language that targets System FO to opt out of the guardrails provided where self-application patterns are demanded by the programmer.

Practical and engineering-focused avenues of future research include investigating the practical considerations necessary to apply the work of System FO to compiler optimisations to enable long-running decidable optimisation processes. Moreover, investigation is warranted into long-running automated proof search. Additionally, where the work of Chapter 5 presents an online theory of termination checking, it is likely to be adaptable to static termination checking similar to a sized-types based approach. Given the language describes a WQO on the entire term structure of the program, future work is warranted in formalising static size-change termination analysis within System FO.

In short, System FO presents numerous further research avenues in various fields, including programming language design, type theory, compiler optimisations and proof search.

Chapter 7

Conclusion

This thesis has presented an exploration of the design space for the implementation of unsaturated type families, culminating in a theoretically sound and practically robust implementation. In the process, this thesis presents an alternative formal specification from that of which is proposed in earlier works (Kiss 2018; Eisenberg 2016) that may also soundly and elegantly support unsaturated type families.

Next, this thesis both explores the design space of theoretical specifications and highlights the underlying tension between theoretical soundness and practical implementability and leverages class constraints as an evidence passing mechanism. The final implementation takes inspiration from early works regarding unsaturated type families and proves sound where early approaches failed. The final theoretical specification for the proposed solution is adapted those early works (Kiss 2018, Chen 2024), while unifying provenance-based defaulting rules into a single structural check. This unification is key; matchability as a kind arrow annotation is an annotation regarding semantics, not provenance, and therefore should be treated as such.

In exploring the engineering considerations for alternative approaches, this thesis details the architectural work necessary to fully support arbitrary elimination of trivial boxed constructs within Haskell as a particularly useful optimisation pass. The erasure of trivial boxed data relies on both erasing constructs by type within code generation, and the adequate modification of memory representations of data constructors. This thesis identifies the simplifier pass as the ideal resting place for such an optimisation.

This thesis has also presented System FO, a decidable yet expressive adaptation of System F that marks, to the knowledge of this thesis, the first decidable typed variation of lambda calculus that retains a fixed point combinator. System FO dramatically improves the scope of decidability in comparison to traditional size-change termination approaches. By applying embedding-based divergence detection onto a higher-order language that is stratified to establish a well-quasi order on terms, this thesis achieves what has often been considered unreconcilable: decidability in the presence of a fixed point combinator. In turn, the termination analysis of programs that admit fixed point combinators under this system is no longer undecidable and is instead a question of computational complexity.

Both of these contributions mark progress toward enabling more powerful, expressive and decidable type-level computation and are a step forward toward a fully Dependent Haskell.

Bibliography

Abel, Andreas (Dec. 2010). "MiniAgda: Integrating Sized and Dependent Types". In: *Electronic Proceedings in Theoretical Computer Science* 43, pp. 14–28. ISSN: 2075-2180. DOI: 10.4204/eptcs.43.2. URL: http://dx.doi.org/10.4204/EPTCS.43.2.

ABEL, ANDREAS and BRIGITTE PIENTKA (2016). "Well-founded recursion with copatterns and sized types". In: *Journal of Functional Programming* 26, e2. DOI: 10.1017/S0956796816000022.

Berthelette, Sophie, Gilles Brassard, and Xavier Coiteux-Roy (June 2024). "On computable numbers, with an application to the Druckproblem". In: *Theor. Comput. Sci.* 1002.C. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2024.114573. URL: https://doi.org/10.1016/j.tcs.2024.114573.

Chen, Yitang (2024). "Implementing Unsaturated Type Families in Haskell". Thesis not publicly available. MA thesis. Imperial College London.

Coquand, Thierry and Gerard Huet (Feb. 1988). "The calculus of constructions". In: *Inf. Comput.* 76.2–3, pp. 95–120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3. URL: https://doi.org/10.1016/0890-5401(88)90005-3.

Dershowitz, Nachum (1987). "Termination of rewriting". In: *Journal of Symbolic Computation* 3.1, pp. 69–115. ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(87)80022-6. URL: https://www.sciencedirect.com/science/article/pii/S0747717187800226.

— (1993). "Trees, ordinals and termination". In: *TAPSOFT'93: Theory and Practice of Software Development.* Ed. by M. -C. Gaudel and J. -P. Jouannaud. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 243–250. ISBN: 978-3-540-47598-9.

Eisenberg, Richard A. (2015). *System FC, as Implemented in GHC.* Tech. rep. MS-CIS-15-09. Department of Computer and Information Science, University of Pennsylvania. URL: https://repository.brynmawr.edu/cgi/viewcontent.cgi?article=1015&context=compsci_pubs.

— (2016). "Dependent Types in Haskell: Theory and Practice". In: *CoRR* abs/1610.07978. arXiv: 1610.07978. URL: http://arxiv.org/abs/1610.07978.

Eisenberg, Richard A. and Stephanie Weirich (2012). "Dependently typed programming with singletons". In: *Proceedings of the 2012 Haskell Symposium.* Haskell '12. Copenhagen, Denmark: Association for Computing Machinery, pp. 117–130. ISBN: 9781450315746. DOI: 10.1145/2364506.2364522. URL: https://doi.org/10.1145/2364506.2364522.

Girard, Jean-Yves (1972). "Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur". In: URL: https://api.semanticscholar.org/CorpusID:117631778.

Gundry, Adam Michael (2013). "Type inference, Haskell and dependent types". PhD thesis. University of Strathclyde, Glasgow, UK. URL: http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full%5C&object%5C_id=22728.

Hinze, Ralf (July 2000). “Generalizing generalized tries”. In: *J. Funct. Program.* 10.4, pp. 327–351. ISSN: 0956-7968. DOI: 10.1017/S0956796800003713. URL: https://doi.org/10.1017/S0956796800003713.

— (Apr. 2003). “Fun with phantom types”. In.

Jhala, Ranjit (Jan. 2014). “Refinement types for Haskell”. In: pp. 27–27. DOI: 10.1145/2541568.2541569.

Jones, Simon L. Peyton, Geoffrey Washburn, and Stephanie Weirich (2004). “Wobbly types: type inference for generalised algebraic data types”. In: URL: https://api.semanticscholar.org/CorpusID:16021198.

Kiss, Csongor (2018). “Higher-Order Type-Level Programming in Haskell”. Available online at https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-ug-projects/Csongor-Kiss-Higher-order-type-level-programming-in-Haskell.pdf. MA thesis. Imperial College London. URL: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1718-ug-projects/Csongor-Kiss-Higher-order-type-level-programming-in-Haskell.pdf.

Kruskal, J. B. (1960). “Well-Quasi-Ordering, The Tree Theorem, and Vazsonyi’s Conjecture”. In: *Transactions of the American Mathematical Society* 95.2, pp. 210–225. ISSN: 00029947. URL: http://www.jstor.org/stable/1993287.

Leuschel, Michael (2002). “Homeomorphic Embedding for Online Termination of Symbolic Methods”. In: *The Essence of Computation - Essays dedicated to Neil Jones.* Ed. by Torben Mogensen, David Schmidt, and I. H. Sudborough. Springer, pp. 379–403. URL: https://eprints.soton.ac.uk/257252/.

Loader, Ralph (2001). “Finitary PCF is not decidable”. In: *Theoretical Computer Science* 266.1, pp. 341–364. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(00)00194-8. URL: https://www.sciencedirect.com/science/article/pii/S0304397500001948.

Marlow, Simon et al. (2010). *Haskell 2010 Language Report.* https://www.haskell.org/onlinereport/haskell2010/. [Online; accessed June 2, 2025].

Martin-Löf, Per (1980). *Intuitionistic Type Theory.*

Meijer, Erik, Maarten Fokkinga, and Ross Paterson (1991). “Functional programming with bananas, lenses, envelopes and barbed wire”. In: *Functional Programming Languages and Computer Architecture.* Ed. by John Hughes. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 124–144. ISBN: 978-3-540-47599-6.

Peyton Jones, Simon et al. (Jan. 2007). “Practical type inference for arbitrary-rank types”. In: *J. Funct. Program.* 17.1, pp. 1–82. ISSN: 0956-7968. DOI: 10.1017/S0956796806006034. URL: https://doi.org/10.1017/S0956796806006034.

Plotkin, G.D. (1977). “LCF considered as a programming language”. In: *Theoretical Computer Science* 5.3, pp. 223–255. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(77)90044-5. URL: https://www.sciencedirect.com/science/article/pii/0304397577900445.

Reynolds, John C. (1972). “Definitional interpreters for higher-order programming languages”. In: *Proceedings of the ACM Annual Conference - Volume 2.* ACM ’72. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852. URL: https://doi.org/10.1145/800194.805852.

— (1974). “Towards a theory of type structure”. In: *Programming Symposium, Proceedings Colloque Sur La Programmation.* Berlin, Heidelberg: Springer-Verlag, pp. 408–423. ISBN: 3540068597.

Schrijvers, Tom et al. (Sept. 2008). “Type checking with open type functions”. In: vol. 43, pp. 51–62. DOI: 10.1145/1411204.1411215.

Sørensen, Morten, R. Glück, and Neil Jones (Nov. 1996). “A positive supercompiler”. In: *Journal of Functional Programming* 6, pp. 811–838. DOI: 10.1017/S0956796800002008.

Spiwack, Arnaud (2018). *Linear Types: A GHC Proposal.* https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0111-linear-types.rst.

The GHC Team (2025). *GHC User's Guide.* Accessed June 2, 2025. The GHC Team. URL: https://downloads.haskell.org/ghc/latest/docs/users_guide/.

Vytiniotis, Dimitrios et al. (Sept. 2011). “OutsideIn(X): Modular type inference with local assumptions”. In: *Journal of Functional Programming* 21, pp. 333–412. URL: https://www.microsoft.com/en-us/research/publication/outsideinx-modular-type-inference-with-local-assumptions/.

Weirich, Stephanie, Justin Hsu, and Richard A. Eisenberg (2013). “System FC with explicit kind equality”. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.* ICFP ’13. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 275–286. ISBN: 9781450323260. DOI: 10.1145/2500365.2500599. URL: https://doi.org/10.1145/2500365.2500599.

Weirich, Stephanie, Antoine Voizard, et al. (Aug. 2017). “A specification for dependent types in Haskell”. In: *Proc. ACM Program. Lang.* 1.ICFP. DOI: 10.1145/3110275. URL: https://doi.org/10.1145/3110275.

Yang, Zhixuan and Nicolas Wu (2022). “Fantastic Morphisms and Where to Find Them: A Guide to Recursion Schemes”. In: *Mathematics of Program Construction: 14th International Conference, MPC 2022, Tbilisi, Georgia, September 26–28, 2022, Proceedings.* Tblilisi, Georgia: Springer-Verlag, pp. 222–267. ISBN: 978-3-031-16911-3. DOI: 10.1007/978-3-031-16912-0_9. URL: https://doi.org/10.1007/978-3-031-16912-0_9.

Yorgey, Brent A. et al. (2012). “Giving Haskell a Promotion”. In: *Proceedings of the Haskell Symposium 2012.* ACM. DOI: 10.1145/2103786.2103795. URL: https://doi.org/10.1145/2103786.2103795.