

# Chapter 1

## Introduction

I have formatted this proposal as an outline of my eventual dissertation, with drafts of parts of the final text already written. Paragraphs rendered in blue, such as this one, are meant to appear only in the proposal.

Haskell has become a wonderful ~~and rich~~ playground for type system experimentation. Despite its relative longevity – at roughly 25 years old [26] – type theorists still turn to Haskell as a place to build new type system ideas and see how they work in a practical setting [3, 7–9, 18, 24, 27–29, 31]. As a result, Haskell’s type system has grown ever more intricate over the years. As the power of types in Haskell has increased, Haskellers have started to integrate dependent types into their programs [2, 16, 32, 34], despite the fact that today’s Haskell<sup>1</sup> does not internally support dependent types. Indeed, the desire to program in Haskell but with support for dependent types influenced the creation of Agda [36] and Idris [5]; both are Haskell-like languages with support for full dependent types. I draw comparisons between my work and these two languages, as well as Coq [10], throughout **this dissertation**.

**This dissertation** closes the gap, by adding support for dependent types into Haskell. In this work, I **detail both** the changes to GHC’s internal language, known as System FC [44], **and explain** the changes to the surface language necessary to support dependent types. Naturally, I must also describe the elaboration from the surface language to the internal language, including type inference. Along with the textual description contained in this dissertation, I have also implemented support for dependent types in GHC directly; I expect a future release of the software to include this support. Much of my work builds upon the critical work of Gundry [21]; one of my chief contributions is adapting his work to work with the GHC implementation and further features of Haskell.

Specifically, I offer the following contributions:

- Chapter 3 includes a series of examples of dependently typed programming in Haskell. Though a fine line is hard to draw, these examples are divided into two

---

<sup>1</sup>Throughout this dissertation, a reference to “today’s Haskell” refers to the language implemented by the Glasgow Haskell Compiler (GHC), version 7.10, released in 2015.

categories: programs where rich types give a programmer more compile-time checks of her algorithms, and programs where rich types allow a programmer to express a more intricate algorithm that may not be well-typed under a simpler system. Section 3.3 then argues why dependent types in Haskell, in particular, are an interesting and worthwhile subject of study.

Although no new results, as such, are presented in Chapter 3, gathering these examples together is a true contribution of this dissertation. At the time of writing, dependent types are still rather esoteric in the functional programming community, and examples of how dependent types can do real work (outside of theorem-proving, which is beyond the scope of dependent types in Haskell – see Section 3.3.3) are hard to come by.

- Chapter 4 is a thorough treatment of System FC, as it has inspired today’s GHC. Though there are many publications on System FC [7, 18, 44, 52, 53], it has evolved much over the years and benefits from a solid, full treatment. Having a record of today’s System FC also allows readers to understand the extensions I add in context.

This chapter, however, does not prove type safety of this language, deferring the proof to the system that includes dependent types.

- Chapter 5 presents Dependent Haskell, the language I have designed in this dissertation. This chapter is written to be useful to practitioners, being a user manual of sorts of the new features. In combination with Chapter 3, this chapter could serve to educate Haskellers on how to use the new features.
- Chapter 6 describes System FC with dependent types, which I have named System FCD. System FCD is an extension on System FC, with two major changes:
  1. System FCD supports first-class equalities among the kinds that classify the system’s types, whereas System FC supports only *type* equalities. Adding kind equalities is made simpler by also adding the *Type-in-Type* axiom (or  $\star :: \star$ ) and merging the grammar of types and kinds. This aspect of System FCD was originally presented in the work of Weirich et al. [52].
  2. System FCD also contains a proper  $\Pi$ -type, which quantifies over an argument **made available in both a type and a term**. It is the existence of  $\Pi$ -types that enables me to claim that the language is dependently typed.

This chapter contains the **full prose** and technical description of System FCD and well as a proof of type safety (though the details of many proofs are relegated to Appendix B).

- Chapter 7 introduces a specification of the Dependent Haskell surface language. Though much formal work has been done on System FC – the *internal* language



— there is much less formal work done on Haskell itself. This chapter builds a specification of the surface language, to be used when discussing type inference and elaboration into System FCD.

- Chapter 8 presents the type inference and elaboration algorithm from Dependent Haskell to System FCD. As compared to Gundry’s work [21], the chief novelty here is that it adapts the type inference algorithm to work with (a slight variant of) the OUTSIDEIN algorithm [49]. This chapter contains proofs of soundness with respect both to the Haskell specification and System FCD. The inference algorithm is not complete, however, though I do prove completeness for subsets of Haskell. This lack of completeness follows directly from the lack of completeness of OUTSIDEIN. See Section 8.3.1 for more details.
- Chapter 9 considers some of the implementation challenges inherent in building Dependent Haskell for wide distribution.
- Chapter 10 puts this work in context by comparing it to several other dependently typed systems, both theories and implementations.

Though not a new contribution, Chapter 2 contains a review of features available in today’s Haskell that support dependently typed programming. This is included as a primer to these features for readers less experienced in Haskell, and also as a counterpoint to the features discussed as parts of Dependent Haskell.

Chapter 11 contains a [timeline of the work remaining toward completing this dissertation](#).

With an implementation of dependent types in Haskell available, I look forward to seeing how the Haskell community builds on top of my work and discovers more and more applications of dependent types.

The type of *smooshList* says that it can be called at any type *a*, as long as there exists an instance *Show a*. The body of *smooshList* can then make use of the *Show a* constraint by calling the *show* method. If we leave out the *Show a* constraint, then the call to *show* does not type-check. This is a direct result of the fact that the full type of *show* is really *Show a*  $\Rightarrow$  *a*  $\rightarrow$  *String*. (The *Show a* constraint on *show* is implicit, as the method is declared within the *Show* class declaration.) Thus, we need to know that the instance *Show a* exists before calling *show* at type *a*.

Operationally, type classes work by passing *dictionaries* [23]. A type class dictionary is simply a record containing all of the methods defined in the type class. It is as if we had these definitions:

```
data ShowDict a = MkShowDict { showMethod :: a  $\rightarrow$  String }
showBool :: Bool  $\rightarrow$  String
showBool True = "True"
showBool False = "False"
showDictBool :: ShowDict Bool
showDictBool = MkShowDict showBool
```

Then, whenever a constraint *Show Bool* must be satisfied, GHC produces the *showDictBool* dictionary. This dictionary actually becomes a runtime argument to functions with a *Show* constraint. Thus, in a running program, the *smooshList* function actually takes 2 arguments: the dictionary corresponding to *Show a* and the list [*a*].

## 2.2 Families

### 2.2.1 Type families

A *type family* [8, 9, 18] is simply a function on types. (I sometimes use “type function” and “type family” interchangeably.) Here is an uninteresting example:

```
type family F1 a where
  F1 Int    = Bool
  F1 Char   = Double
useF1 :: F1 Int  $\rightarrow$  F1 Char
useF1 True  = 1.0
useF1 False = (-1.0)
```

We see that GHC simplifies *F<sub>1</sub> Int* to *Bool* and *F<sub>1</sub> Char* to *Double* in order to typecheck *useF<sub>1</sub>*.

*F<sub>1</sub>* is a *closed* type family, in that all of its defining equations are given in one place. This most closely corresponds to what functional programmers expect from their functions. Today’s Haskell also supports *open* type families, where the set of defining equations can be extended arbitrarily. Open type families interact particularly well

with Haskell's type classes, which can also be extended arbitrarily. Here is a more interesting example than the one above:

```
type family Element c
class Collection c where
    singleton :: Element c → c
type instance Element [a] = a
instance Collection [a] where
    singleton x = [x]
type instance Element (Set a) = a
instance Collection (Set a) where
    singleton = Set.singleton
```

Because the type family *Element* is open, it can be extended whenever a programmer creates a new collection type.

Often, open type families are extended in close correspondence with a type class, as we see here. For this reason, GHC supports *associated* open type families, using this syntax:

```
class Collection' c where
    type Element' c
    singleton' :: Element' c → c
instance Collection' [a] where
    type Element' [a] = a
    singleton' x = [x]
instance Collection' (Set a) where
    type Element' (Set a) = a
    singleton' = Set.singleton
```

Associated type families are essentially syntactic sugar for regular open type families.

**Partiality in type families** A type family may be *partial*, in that it is not defined over all possible inputs. This poses no **direct** problems in the theory or practice of type families. If a type family is used at a type for which it is not defined, the type family application is considered to be *stuck*. For example:

```
type family F2 a
type instance F2 Int = Bool
```

Suppose there are no further instances of *F*<sub>2</sub>. Then, the type *F*<sub>2</sub> *Char* is stuck. It does not evaluate, and is equal only to itself.

Because type family applications cannot be used on the left-hand side of type family equations, it is impossible for a Haskell program to detect whether or not a type is stuck. This is correct behavior, because a stuck open type family might become unstuck with the inclusion of more modules, defining more type family instances.

## 2.2.2 Data families

A *data family* defines a family of datatypes. An example shows best how this works:

```
data family Array a  -- compact storage of elements of type a
data instance Array Bool = MkArrayBool ByteArray
data instance Array Int  = MkArrayInt  (Vector Int)
```

With such a definition, we can have a different runtime representation for an *Array Bool* as we do for an *Array Int*, something not possible with more traditional parameterized types.

Data families do not play a large role in this dissertation.

## 2.3 Rich kinds

### 2.3.1 Kinds in Haskell98

With type families, we can write type-level programs. But are our type-level programs correct? We can gain confidence in the correctness of the type-level programs by ensuring that they are well-kinded. Indeed, GHC does this already. For example, if we try to say *Element Maybe*, we get a type error saying that the argument to *Element* should have kind  $\star$ , but *Maybe* has kind  $\star \rightarrow \star$ .

Kinds in Haskell are not a new invention; they are mentioned in the Haskell98 report [39]. Because type constructors in Haskell may appear without their arguments, Haskell needs a kinding system to keep all the types in line. For example, consider the library definition of *Maybe*:

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

The word *Maybe*, all by itself, does not really represent a type. *Maybe Int* and *Maybe Bool* are types, but *Maybe* is not. The type-level constant *Maybe* needs to be given a type to become a type. The kind-level constant  $\star$  contains proper types, like *Int* and *Bool*. Thus, *Maybe* has kind  $\star \rightarrow \star$ .

Accordingly, Haskell's kind system accepts *Maybe Int* and *Element [Bool]*, but rejects *Maybe Maybe* and *Bool Int* as ill-kinded.

### 2.3.2 Promoted datatypes

The kind system in Haskell98 is rather limited. It is generated by the grammar  $\kappa ::= \star \mid \kappa \rightarrow \kappa$ , and that's it. When we start writing interesting type-level programs, this almost-unity-typed limitation bites.

Accordingly, Yorgey et al. [53] introduce promoted datatypes. The central idea behind promoted datatypes is that when we say

A kind-polymorphic type has extra, invisible parameters that correspond to kind arguments. When I say *invisible* here, I mean that the arguments do not appear in Haskell source code. With the `-fprint-explicit-kinds` flag, GHC will print kind parameters when they occur. Thus, if a Haskell program contains the type `T Maybe Bool` and GHC needs to print this type with `-fprint-explicit-kinds`, it will print `T * Maybe Bool`, making the `*` kind parameter visible. Today's Haskell makes an inflexible choice that kind arguments are always invisible, which is relaxed in Dependent Haskell. See Section 5.1.3 for more information on visibility in Dependent Haskell and Section 5.6.2 for more information on visible kind arguments.

### 2.3.4 Constraint kinds

Yorgey et al. [53] introduces one final extension to Haskell: constraint kinds. Haskell allows constraints to be given on types. For example, the type `Show a => a -> String` classifies a function that takes one argument, of type `a`. The `Show a =>` constraint means that `a` is required to be a member of the `Show` type class. Constraint kinds make constraints fully first-class. We can now write the kind of `Show` as `* -> Constraint`. That is, `Show Int` (for example) is of kind `Constraint`. `Constraint` is a first-class kind, and can be quantified over. A useful construct over `Constraints` is the `Some` type:

```
data Some :: (* -> Constraint) -> * where
  Some :: c a => a -> Some c
```

If we have a value of `Some Show`, stored inside it must be a term of some (existentially quantified) type `a` such that `Show a`. When we pattern-match against the constructor `Some`, we can use this `Show a` constraint. Accordingly, the following function type-checks (where `show :: Show a => a -> String` is a standard library function):

```
showSomething :: Some Show -> String
showSomething (Some thing) = show thing
```

Note that there is no `Show a` constraint in the function signature – we get the constraint from pattern-matching on `Some`, instead.

The type `Some` is useful if, say, we want a heterogeneous list such that every element of the list satisfies some constraint. That is, `[Some Show]` can have elements of any type `a`, as long as `Show a` holds:



```
heteroList :: [Some Show]
heteroList = [Some True, Some (5 :: Int), Some (Just ())]
printList :: [Some Show] -> String
printList things = "[" ++ intercalate ", " (map showSomething things) ++ "]"
```

```
λ> putStrLn $ printList heteroList
[ True, 5, Just ()]
```

## 2.4 Generalized algebraic datatypes

Generalized algebraic datatypes (or GADTs) are a powerful feature that allows term-level pattern matches to refine information about types. They undergird much of the programming we will see in the examples in Chapter 3, and so I defer most of the discussion of GADTs to that chapter.

Here, I introduce one particularly important GADT: propositional equality. The following definition appears now as part of the standard library shipped with GHC, in the *Data.Type.Equality* module:

```
data (a :: k) :~: (b :: k) where  
  Refl :: a :~: a
```

The idea here is that a value of type  $\tau :~: \sigma$  (for some  $\tau$  and  $\sigma$ ) represents evidence that the type  $\tau$  is in fact equal to the type  $\sigma$ . Here is a (trivial) use of this type, also from *Data.Type.Equality*:

```
castWith :: (a :~: b) → a → b  
castWith Refl x = x
```

Here, the *castWith* function takes a term of type  $a :~: b$  – evidence that *a* equals *b* – and a term of type *a*. It can immediately return this term, *x*, because GHC knows that *a* and *b* are the same type. Thus, *x* also has type *b* and the function is well typed.

Note that *castWith* must pattern-match against *Refl*. The reason this is necessary becomes more apparent if we look at an alternate, entirely equivalent way of defining ( $:~:$ ):

```
data (a :: k) :~: (b :: k) =  
  (a ~ b) ⇒ Refl
```

In this variant, I define the type using the Haskell98-style syntax for datatypes. This says that the *Refl* constructor takes no arguments, but does require the constraint that  $a \sim b$ . The constraint ( $\sim$ ) is GHC's notation for a proper type equality constraint. Accordingly, to use *Refl* at a type  $\tau :~: \sigma$ , GHC must know that  $\tau \sim \sigma$  – in other words, that  $\tau$  and  $\sigma$  are the same type. When *Refl* is matched against, this constraint  $\tau \sim \sigma$  becomes available for use in the body of the pattern match.

Returning to *castWith*, pattern-matching against *Refl* brings  $a \sim b$  into the context, and GHC can apply this equality in the right-hand side of the equation to say that *x* has type *b*.

Operationally, the pattern-match against *Refl* is also important. This match is what forces the equality evidence to be reduced to a value. As Haskell is a lazy language, it is possible to pass around equality evidence that is  $\perp$ . Matching evaluates the argument, making sure that the evidence is real. The fact that type equality evidence must exist and be executed at runtime is somewhat unfortunate. See Section 3.3.3 for some discussion.



## 2.5 Higher-rank types

Standard ML and Haskell98 both use, essentially, the Hindley-Milner (HM) type system [11, 25, 35]. The HM type system allows only *prenex quantification*, where a type can quantify over type variables only at the very top. The system is based on *types*, which have no quantification, and *type schemes*, which do:

$$\begin{aligned}\tau &::= \alpha \mid H \mid \tau_1 \ \tau_2 && \text{types} \\ \sigma &::= \forall \alpha. \sigma \mid \tau && \text{type schemes}\end{aligned}$$

Here, I use  $\alpha$  to stand for any of a countably infinite set of type variables and  $H$  to stand for any type constant (including  $(\rightarrow)$ ).

Let-bound definitions in HM are assigned type schemes; lambda-bound definitions are assigned monomorphic types, only. Thus, in HM, it is appropriate to have a function `length ::  $\forall a. [a] \rightarrow \text{Int}$`  but disallowed to have one like `bad ::  $(\forall a. a \rightarrow a \rightarrow a) \rightarrow \text{Int}$` : `bad`'s type has a  $\forall$  somewhere other than at the top of the type. This type is of the second rank, and is forbidden in HM.

On the other hand, today's GHC allows types of arbitrary rank. Though a full example of the usefulness of this ability would take us too far afield, Lämmel and Peyton Jones [31] and Washburn and Weirich [51] (among others) make critical use of this ability. The cost, however, is that higher-rank types cannot be inferred. For this reason, the following code

`higherRank x = (x True, x (5 :: Int))` ... But is HAS a type signature in it... Confusing...

will not compile without a type signature. Without the signature, GHC tries to unify the types `Int` and `Bool`, failing. However, providing a signature

`higherRank ::  $(\forall a. a \rightarrow a) \rightarrow (\text{Bool}, \text{Int})$`

does the trick nicely.

Type inference in the presence of higher-rank types is well studied, and can be made practical via bidirectional type-checking [13, 40].

## 2.6 Scoped type variables

A modest, but subtle, extension in GHC is `ScopedTypeVariables`, which allows a programmer to refer back to a declared type variable from within the body of a function. As dealing with scoped type variables can be a point of confusion for Haskell type-level programmers, I include a discussion of it here.

Consider this implementation of the left fold `foldl`:

`foldl ::  $(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$`   
`foldl f z0 xs0 = lgo z0 xs0`

no two instances where the same choice for  $a$  but differing choices for  $b$  could be made.

Functional dependencies are, in some ways, more powerful than type families. For example, consider this definition of *Plus*:

```
class Plus (a :: Nat) (b :: Nat) (r :: Nat) | a b → r, r a → b
instance Plus 'Zero b b
instance Plus a b r ⇒ Plus ('Succ a) b ('Succ r)
```

The functional dependencies for *Plus* are more expressive than what we can do for type families. (However, see the work of Stolarek et al. [43], which attempts to close this gap.) They say that  $a$  and  $b$  determine  $r$ , just like the arguments to a type family determine the result, but also that  $r$  and  $a$  determine  $b$ . Using this second declared functional dependency, if we know *Plus*  $a$   $b$   $r$  and *Plus*  $a$   $b'$   $r$ , we can conclude  $b = b'$ . Although the functional dependency  $r\ b \rightarrow a$  also holds, GHC is unable to prove this.

Functional dependencies have enjoyed a rich history of aiding type-level programming [30, 33, 38]. Yet, they require a different paradigm to much of functional programming. When writing term-level definitions, functional programmers think in terms of functions that take a set of arguments and produce a result. Functional dependencies, however, encode type-level programming through relations, not proper functions. Though both functional dependencies and type families have their proper place in the Haskell ecosystem, I have followed the path taken by other dependently typed languages and use type-level functions as the main building blocks of Dependent Haskell, as opposed to functional dependencies.

# Chapter 3

## Motivation

Functional programmers use dependent types in two primary ways, broadly speaking: in order to eliminate erroneous programs from being accepted, and in order to write intricate programs that a simply-typed language cannot accept. In this chapter, we will motivate the use of dependent types from both of these angles. The chapter concludes with a section motivating why Haskell, in particular, is ripe for dependent types.

As a check for accuracy in these examples and examples throughout this dissertation, all the indented, typeset code is type-checked against my implementation every time the text is typeset.

In this proposal, I elide the details of some of the motivating examples. Instead, I list them as stubs to be filled out later, when writing the dissertation proper.

The code snippets throughout this proposal are presented on a variety of background colors. A light green background highlights code that does not work in today's Haskell but does currently (May 2015) work in my implementation. A light yellow background indicates code that does not work verbatim in my implementation, but could still be implemented via the use of singletons [16] and similar workarounds. A light red background marks code that does not currently work in my implementation due to bugs and incompleteness in my implementation. To my knowledge, there is nothing more than engineering (and perhaps the use of singletons) to get these examples working.

With all of these supporting functions written, the evaluator itself is dead simple:


```

eval (Var v)      = Explain... case v of { } -- no variables in an empty context
eval (Lam body)   = LamVal body
eval (App e1 e2)  = eval (apply (eval e1) e2)
eval TT          = TTVal

```

### 3.1.2.3 Small-step stepper

We now turn to writing the small-step semantics. We could proceed in a very similar fashion to the big-step semantics, by defining a *step* function that steps an expression either to another expression or to a value. But we want better than this.

Instead, we want to ensure that the small-step semantics respects the big-step semantics. That is, after every step, we want the value – as given by the big-step semantics – to remain the same. We thus want the small-step stepper to return a custom datatype, marrying the result of stepping with evidence that the value of this result agrees with the value of the original expression: 

```

data StepResult :: Expr '[ ] ty → ★ where
  Stepped :: Π (e' :: Expr '[ ] ty) → ( 'eval e ~ 'eval e' ) ⇒ StepResult e
  Value   :: Π (v :: Val      ty) → ( 'eval e ~ v )           ⇒ StepResult e

```

A *StepResult e* is the result of stepping an expression *e*. It either contains a new expression *e'* whose value *equals e's value*, or it contains the value *v* that is the result of evaluating *e*.   
<sup>In what sense?</sup>

An interesting detail about these constructors is that they feature an equality constraint *after* a runtime argument. Currently, GHC requires that all data constructors take a sequence of type arguments, followed by constraints, followed by regular arguments. Generalizing this form does not *provide* any real difficulty, however.

With this in hand, the *step* function is remarkably easy to write:

```

step :: Π (e :: Expr '[ ] ty) → StepResult e
step (Var v)      = case v of { } -- no variables in an empty context
step (Lam body)   = Value (LamVal body)
step (App e1 e2)  = case step e1 of
  Stepped e1' → Stepped (App e1' e2)
  Value v     → Stepped (apply v e2)
step TT          = Value TTVal

```

Due to GHC's ability to freely use *assumed equality assumptions*, *step* requires no explicit manipulation of equality proofs. Let's look at the *App* case above. We

Reflection of this sort has been possible for some time using the *Typeable* mechanism [31]. However, the lack of kind equalities – the ability to learn about a type’s kind via pattern matching – has hindered some of the usefulness of Haskell’s reflection facility. In this section, we explore how this is the case and how the problem is fixed.

### 3.2.2.1 Heterogeneous propositional equality

Kind equalities allow for the definition of *heterogeneous propositional equality*, a natural extension to the propositional equality described in Section 2.4:

```
data (a :: k1) :≈:(b :: k2) where
  HRefl :: a :≈: a
```

Pattern-matching on a value of type  $a :≈: b$  to get *HRefl*, where  $a :: k_1$  and  $b :: k_2$ , tells us both that  $k_1 \sim k_2$  and that  $a \sim b$ . As we’ll see below, this more powerful form of equality is essential in building the typed reflection facility we want.

### 3.2.2.2 Type representation

Here is our desired representation:

```
data TyCon (a :: k)
  -- abstract; Int is represented by a TyCon Int .
data TypeRep (a :: k) where
  TyCon :: TyCon a → TypeRep a
  TyApp :: TypeRep a → TypeRep b → TypeRep (a b)
```

For every new type declared, the compiler would supply an appropriate value of the *TyCon* datatype. The type representation library would supply also the following function, which computes equality over *TyCons*, returning the heterogeneous equality witness:

```
eqTyCon :: ∀ (a :: k1) (b :: k2).
  TyCon a → TyCon b → Maybe (a :≈: b)
```

It is critical that this function returns  $(:≈:)$ , not  $(:\sim:)$ . This is because *TyCons* exist at many different kinds. For example, *Int* is at kind  $\star$ , and *Maybe* is at kind  $\star \rightarrow \star$ . Thus, when comparing two *TyCon* representations for equality, we want to learn whether the types *and the kinds* are equal. If we used  $(:\sim:)$  here, then the *eqTyCon* could be used only when we know, from some other source, that the kinds are equal.

We can now easily write an equality test over these type representations:

```

eqT :: ∀ (a :: k1) (b :: k2).
      TypeRep a → TypeRep b → Maybe (a ≈ b)
eqT (TyCon t1) (TyCon t2) = eqTyCon t1 t2
eqT (TyApp a1 b1) (TyApp a2 b2)
  | Just HRefl ← eqT a1 a2
  , Just HRefl ← eqT b1 b2      = Just HRefl
eqT _ _                        = Nothing

```

Note the extra power we get by returning *Maybe*  $(a \approx b)$  instead of just a *Bool*. When the types indeed equal, we get evidence that GHC can use to **be away** of this type equality during type-checking. A simple return type of *Bool* would not give the type-checker any information.

### 3.2.2.3 Dynamic typing

Now that we have a type representation with computable equality, we can package that representation with a chunk of data, and so form a dynamically typed package:

```

data Dyn where
  Dyn :: ∀ (a :: ★). TypeRep a → a → Dyn

```

The *a* type variable there is an *existential* type variable. We can think of this type as being part of the data payload of the *Dyn* constructor; it is chosen at construction time and unpacked at pattern-match time. Because of the *TypeRep a* argument, we can learn more about *a* after unpacking. (Without the *TypeRep a* or some other type-level information about *a*, the unpacking code must treat *a* as an unknown type and must be parametric in the choice of type for *a*.)

Using *Dyn*, we can pack up arbitrary data along with its type, and push that data across a network. The receiving program can then make use of the data, without needing to subvert Haskell’s type system. The type representation library must be trusted to recreate the *TypeRep* on the far end of the wire, but the equality tests above and other functions below can live outside the trusted code base.

Suppose we were to send an object with a function type, say *Bool* → *Int* over the network. For the time being, let’s ignore the complexities of actually serializing a function – there is a solution to that problem<sup>3</sup>, but here we are concerned only with the types. We would want to apply the received function to some argument. What we want is this:

```

dynApply :: Dyn → Dyn → Maybe Dyn

```

The function *dynApply* applies its first argument to the second, as long as the types line up. The definition of this function is fairly straightforward:

<sup>3</sup><https://ghc.haskell.org/trac/ghc/wiki/StaticPointers>


Although it is conceivable that dependent types are available only with the new `DependentTypes` extension that is not backward compatible, this is not what I have done. Instead, with two exceptions, all programs that compile without `DependentTypes` continue to compile with that extension enabled. The two exceptions are as follows:

- There is now a parsing ambiguity around the glyph `*`. Today’s Haskell treats `*` in a kind as the kind of types. It is parsed just like an alphanumeric identifier. On the other hand, `*` in types is an infix binary operator with a user-assigned fixity. This leads to an ambiguity: is `Foo * Int` the operator `*` applied to `Foo` and `Int` or is it `Foo` applied to `*` and `Int`? The resolution to this problem is detailed in Section 5.6.1 but it is not backward-compatible in all cases.
- The `DependentTypes` extension implies the `MonoLocalBinds` extension, which disables **let**-generalization when the **let**-bound definition is not closed. This is described in the work of Vytiniotis et al. [48]. This restriction does not bite often in practice, as discussed in the cited work.

Other than these two trouble spots, all programs that compiled previously continue to do so.

The requirement of backward compatibility “keeps me honest” in my design of type inference – I cannot cheat by asking the user for more information. The technical content of this statement is discussed in Chapter 8 by comparison with the work of Vytiniotis et al. [49]. A further advantage of working in Haskell is that the type inference of Haskell is well-studied in the literature. This dissertation continues this tradition in Chapter 8.

### 3.3.3 No termination or totality checking

Existing dependently typed languages strive  the proof systems as well as programming languages. (A notable exception is Cayenne [1], which also omits termination checking, but that language seems to have faded into history.) They care deeply about totality: that all pattern matches consider all possibilities and that every function can be proved to terminate. Coq does not accept a function until it is proved to terminate. Agda behaves likewise, although the termination checker can be disabled on a per-function basis. Idris embraces partiality, but then refuses to evaluate partial functions during type-checking. Dependent Haskell, on the other hand, does not care about totality.

Dependent Haskell emphatically does *not* strive to be a proof system. In a proof system, whether or not a type is inhabited is equivalent to whether or not a proposition holds. Yet, in Haskell, *all* types are inhabited, by  $\perp$  and other looping terms, at a minimum. Even at the type level, all kinds are inhabited by the following type family, defined in GHC’s standard library:

```
type family Any :: k -- no instances
```

# Chapter 5

## Dependent Haskell

This chapter lays out the differences between the Haskell of today and Dependent Haskell. The most important distinction is the introduction of more quantifiers, which we will study first.

### 5.1 Quantifiers



A *quantifier* is a type-level operator that introduces the type of an abstraction, or function. In Dependent Haskell, there are three essential properties of quantifiers, each of which can vary independently of the others. To understand the range of quantifiers that the language offers, we must go through each of these properties. In the text that follows, I use the term *quantifree* to refer to the argument quantified over. The *quantifier body* is the type “to the right” of the quantifier.

#### 5.1.1 Dependency

A quantifier may be either dependent or non-dependent. A dependent quantifree may be used in the quantifier body; a non-dependent quantifree may not.

Today’s Haskell uses  $\forall$  for dependent quantification, as follows:

$$id :: \forall a. a \rightarrow a$$

In this example,  $a$  is the quantifree, and  $a \rightarrow a$  is the quantifier body. Note that  $a$  is used in the quantifier body.

The normal function arrow ( $\rightarrow$ ) is an example of a non-dependent quantifier. Consider the predecessor function:

$$pred :: Int \rightarrow Int$$

The  $Int$  quantifree is not named in the type, nor is it mentioned in the quantifier body.

Dependent Haskell adds a new dependent quantifier,  $\Pi$ , as discussed below.



Quantifier	Dependency	Relevance	Visibility
$\forall (a :: \tau). \dots$	dependent	irrelevant	invisible (unification)
$\forall (a :: \tau) \rightarrow \dots$	dependent	irrelevant	visible
$\Pi (a :: \tau). \dots$	dependent	relevant	invisible (unification)
$\Pi (a :: \tau) \rightarrow \dots$	dependent	relevant	visible
$\tau \Rightarrow \dots$	non-dependent	relevant	invisible (solving)
$\tau \rightarrow \dots$	non-dependent	relevant	visible

Table 5.1: The six quantifiers of Dependent Haskell

to describe visibility; however, these terms are sometimes used in the literature when talking about erasure properties. I will stick to *visible* and *invisible* throughout this dissertation.

Today’s Haskell uses  $(\rightarrow)$  for visible quantification. That is, when we pass an ordinary function an argument, the argument is visible in the Haskell source. For example, the 3 in *pred* 3 is visible.

On the other hand, today’s  $\forall$  and  $(\Rightarrow)$  are invisible quantifiers. When we call *id True*, the *a* in the type of *id* is instantiated at *Bool*, but *Bool* is elided in the call *id True*. During type inference, GHC uses unification to discover that the correct argument to use for *a* is *Bool*.

Invisible arguments specified with  $(\Rightarrow)$  are constraints. Take, for example, *show* ::  $\forall a. \text{Show } a \Rightarrow a \rightarrow \text{String}$ . The *show* function properly takes 3 arguments: the  $\forall$ -quantified type variable *a*, the  $(\Rightarrow)$ -quantified dictionary for *Show a* (see Section 2.1 if this statement surprises you), and the  $(\rightarrow)$ -quantified argument of type *a*. However, we use *show* as, say, *show True*, passing only one argument visibly. The  $\forall a$  argument is discovered by unification to be *Bool*, but the *Show a* argument is discovered using a different mechanism: instance solving and lookup. (See the work of Vytiniotis et al. [49] for the algorithm used.) We thus must be aware that invisible arguments may use different mechanisms for instantiation.

Dependent Haskell offers both visible and invisible forms of  $\forall$  and  $\Pi$ ; the invisible forms instantiate only via unification. Dependent Haskell retains, of course, the invisible quantifier  $(\Rightarrow)$ , which is instantiated via *instanced* lookup and solving. Finally, note that visibility is a quality only of source Haskell. All arguments are always “visible” in System FCD.

#### 5.1.3.1 Invisibility in other languages

#### 5.1.3.2 Visibility overrides

### 5.1.4 The six quantifiers of Dependent Haskell

Now that we have enumerated the quantifier properties, we are ready to describe the six quantifiers that exist in Dependent Haskell. They appear in Table 5.1. The


## 5.3 Inferring $\Pi$

If a function is written at top-level without a signature, will it have any  $\Pi$ -quantified parameters? At the time of writing, I think the answer is “no”. This section will describe inference around  $\Pi$ -quantified parameters and will address how a user-written  $\lambda$ -expression will be treated. (My current plan: all variables bound by an explicit  $\lambda$  will be  $(\rightarrow)$ -quantified, never  $\Pi$ -quantified.)

## 5.4 Shared subset of terms and types

Not every term can appear in a type. For example, today’s Haskell does not permit  $\lambda$ , **case**, or **let** in types. Types also do not yet permit unsaturated type functions. This section will explore the limits of what can be expressed in both types and terms. Depending on time constraints as I write this dissertation, I may work on expanding this subset to include some of the constructs above. Inspiration for doing so will come from previous work [15], which suggests that allowing *universal* promotion may well be possible.

## 5.5 Roles and dependent types

Roles and dependent types  are a tricky interaction, the details of which are beyond the scope of this proposal. One approach to combining the two features appears in a recent draft paper [14]. The user-facing effects of the interaction between roles and dependent types will appear in this section.

## 5.6 Other syntax changes

### 5.6.1 Parsing for $\star$

### 5.6.2 Visible kind variables

Today’s Haskell requires that all kind parameters always be invisible. My work changes this, as will be discussed here.

### 5.6.3 Import and export lists

If any functions are promoted into types, module writers should have the option of whether or not to export a function’s definition along with its type. Exporting just the type will allow the function to be used in terms and in types, but no compile-time reduction will be possible. Exporting the full definition allows, also, compile-time reduction in types. Supporting the choice between these export modes will require a small change to export and import lists, as will be detailed here.

# Chapter 10

## Related work

### 10.1 Comparison to Gundry [21]

### 10.2 Comparison to Idris

Although I compare Dependent Haskell against Coq, Agda, and Idris throughout this proposal, I plan on writing a more detailed comparison against just Idris in this section, as my work is closest to what has been done in Idris, and I think this comparison will be the most illuminating.

### 10.3 Comparison to Liquid Haskell

### 10.4 Applicability beyond Haskell

The knowledge gained in adding dependent types to Haskell will translate to other environments as well. A key example will be the thought of adding dependent types to a variant of ML. Going the other way, I will examine adding more programmatic features to existing dependently typed languages (in particular, Haskell’s **newtype** construct).



### 10.5 Future work

Though this dissertation will deliver  $\Pi$ , that’s not the end of the story. Here are some questions I have that I do not expect will be answered in the course of writing this work.

- How to improve error messages? While my work will strive hard not to degrade error messages for non-dependently-typed programs, I offer no guarantee about the quality of error messages in programs with lots of dependent types. How

5. I will need to write up the dissertation. Given that some of this writing will happen concurrently with achieving the tasks above, I estimate this will take 1 further month.

These estimates total to 9 months. Given that something is sure to fall behind, and that I will take time off from this schedule to write a paper or two during the next year, I think this is all reasonable to complete for late spring of 2016.

Here is a specific timeline of goals and dates:

1. Submit a POPL paper by July 10, 2015.
2. Merge current implementation of kind equalities by Sept. 15, 2015.
3. Write Chapter 6 and Appendix B by Oct. 15, 2015.
4. Implement  $\Pi$  into GHC by Jan. 15, 2016. This implementation will be functional, but perhaps unpolished.
5. Submit an ICFP paper by Mar. 1, 2016.
6. Write Chapter 8 and Appendix C by May 1, 2016.
7. Write remaining parts of dissertation by June 1, 2016.
8. Defend dissertation by July 1, 2016.

Note that I do *not* expect to merge my implementation of Dependent Haskell with GHC's master branch before defending the dissertation. While this would be wonderful to achieve, I do not want to place all of the engineering such a goal would entail on my critical path toward finishing the dissertation. I expect to spend time during the summer of 2016 polishing the implementation and hopefully to merge later in 2016.