DEPENDENT TYPES IN HASKELL: THEORY AND PRACTICE

Richard A. Eisenberg

DISSERTATION PROPOSAL

Contents

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[1] 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

[1] 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 Π-type, which quantifies over an argument made available in both a type and a term. It is the existence of Π-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.

Chapter 2

Preliminaries

This chapter is a primer of type-level programming facilities that exist in today's Haskell. It serves both as a way for readers less experienced in Haskell to understand the remainder of the dissertation, and as a point of comparison against the Dependent Haskell language I describe in Chapter 5. Those more experienced with Haskell may easily skip this chapter. However, all readers may wish to consult Appendix A to learn the typographical conventions used throughout this dissertation.

I assume that the reader is comfortable with a typed functional programming language, such as Haskell98 or a variant of ML.

2.1 Type classes and dictionaries

Haskell supports type classes [50]. An example is worth a thousand words:

```
class Show a where
  show :: a → String
instance Show Bool where
  show True  = "True"
  show False = "False"
```

This declares the class *Show*, parameterized over a type variable *a*, with one method *show*. The class is then instantiated at the type *Bool*, with a custom implementation of *show* for *Bool*s. Note that, in the *Show Bool* instance, the *show* function can use the fact that *a* is now *Bool*: the one argument to *show* can be pattern-matched against *True* and *False*. This is in stark contrast to the usual parametric polymorphism of a function *show'* :: $a \to$ *String*, where the body of *show'* *cannot* assume any particular instantiation for *a*.

With *Show* declared, we can now use this as a constraint on types. For example:

```
smooshList :: Show a ⇒ [a] → String
smooshList xs = concat (map show xs)
```

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 $Show\ a \Rightarrow a \to String$. (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 → String}

showBool :: Bool → 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 → F1 Char
useF1 True  = 1.0
useF1 False = (−1.0)
```

We see that GHC simplifies $F_1\ Int$ to *Bool* and $F_1\ Char$ to *Double* in order to typecheck $useF_1$.

F_1 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_2. Then, the type F_2 *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 $\star \to \star$.

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 $\star \to \star$.

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 $\kappa ::= \star \mid \kappa \to \kappa$, and that's it. When we start writing interesting type-level programs, this almost-unityped limitation bites.

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

```
data Bool = False | True
```

we declare two entities: a type *Bool* inhabited by terms *False* and *True*; and a kind *Bool* inhabited by types '*False* and '*True*.[1] We can then use the promoted datatypes for more richly-kinded type-level programming.

A nice, simple example is type-level addition over promoted unary natural numbers:

```
data Nat = Zero | Succ Nat
type family a + b where
  'Zero   + b = b
  'Succ a + b = 'Succ (a + b)
```

Now, we can say 'Succ 'Zero + 'Succ ('Succ 'Zero) and GHC will simplify the type to 'Succ ('Succ ('Succ 'Zero)). We can also see here that GHC does kind inference on the definition for the type-level +. We could also specify the kinds ourselves like this:

```
data family (a :: Nat) + (b :: Nat) :: Nat where ...
```

Yorgey et al. [53] detail certain restrictions in what datatypes can be promoted. A chief contribution of this dissertation is lifting these restrictions.

2.3.3 Kind polymorphism

A separate contribution of the work of Yorgey et al. [53] is to enable *kind polymorphism*. Kind polymorphism is nothing more than allowing kind variables to be held abstract, just like functional programmers frequently do with type variables. For example, here is a type function that calculates the length of a type-level list at any kind:

```
type family Length (list :: [k]) :: Nat where
  Length '[]       = 'Zero
  Length (x ': xs) = 'Succ (Length xs)
```

Kind polymorphism extends naturally to constructs other than type functions. Consider this datatype:

```
data T f a = MkT (f a)
```

With the `PolyKinds` extension enabled, GHC will infer a most-general kind $\forall\ k.\ (k \to \star) \to k \to \star$ for *T*. In Haskell98, on the other hand, this type would have kind $(\star \to \star) \to \star \to \star$, which is strictly less general.

[1] The new kind does not get a tick ' but the new types do. This is to disambiguate a promoted data constructor 'X from a declared type X; Haskell maintains separate type and term namespaces. The ticks are optional if there is no ambiguity, but I will always use them throughout this dissertation.

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* $\star$ *Maybe Bool*, making the $\star$ 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* $\Rightarrow$ *a* $\rightarrow$ *String* classifies a function that takes one argument, of type *a*. The *Show a* $\Rightarrow$ 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 $\star \rightarrow$ *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 *Constraint*s 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* $\Rightarrow$ *a* $\rightarrow$ *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 $\tau :\sim: \sigma$ (for some τ and σ) represents evidence that the type τ is in fact equal to the type σ. 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 $a :\sim: b$ – 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 $(:\sim:)$:

```
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 $\tau :\sim: \sigma$, GHC must know that $\tau \sim \sigma$ – in other words, that τ and σ 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 $\bot$. 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:

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

Here, I use α 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* :: ∀ *a*. [*a*] → *Int* but disallowed to have one like *bad* :: (∀ *a*. *a* → *a* → *a*) → *Int*: *bad*'s type has a ∀ 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))
```

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 :: (∀ a. a → a) → (Bool, 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 → a → b) → b → [a] → b
foldl f z0 xs0 = lgo z0 xs0
```

```
  where
    lgo z []     = z
    lgo z (x:xs) = lgo (f z x) xs
```

It can be a little hard to see what is going on here, so it would be helpful to add a type signature to the function *lgo*, thus:

```
lgo :: b → [a] → b
```

Yet, doing so leads to type errors. The root cause is that the *a* and *b* in *lgo*'s type signature are considered independent from the *a* and *b* in *foldl*'s type signature. It is as if we've assigned the type *b0* → [*a0*] → *b0* to *lgo*. Note that *lgo* uses *f* in its definition. This *f* is a parameter to the outer *foldl*, and it has type *b* → *a* → *b*. When we call *f z x* in *lgo*, we're passing *z* :: *b0* and *x* :: [*a0*] to *f*, and type errors ensue.

To make the *a* and *b* in *foldl*'s signature be lexically scoped, we simply need to quantify them explicitly. Thus, the following gets accepted:

```
foldl :: ∀ a b. (b → a → b) → b → [a] → b
foldl f z0 xs0 = lgo z0 xs0
  where
    lgo :: b → [a] → b
    lgo z []     = z
    lgo z (x:xs) = lgo (f z x) xs
```

Another particular tricky point around `ScopedTypeVariables` is that GHC will not warn you if you are missing this extension.

2.7 Functional dependencies

Although this dissertation does not dwell much on functional dependencies, I include them here for completeness.

Functional dependencies are GHC's earliest feature introduced to enable rich type-level programming [28, 45]. They are, in many ways, a competitor to type families. With functional dependencies, we can declare that the choice of one parameter to a type class fixes the choice of another parameter. For example:

```
class Pred (a :: Nat) (b :: Nat) | a → b
instance Pred 'Zero     'Zero
instance Pred ('Succ n) n
```

In the declaration for class *Pred* ("predecessor"), we say that the first parameter, *a*, determines the second one, *b*. In other words, *b* has a functional dependency on *a*. The two instance declarations respect the functional dependency, because there are

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 ofStolarek 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 $r\ b \to a$ 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.

3.1 Eliminating erroneous programs

3.1.1 Simple example: Length-indexed vectors

3.1.2 A strongly typed simply typed lambda calculus interpreter

It is straightforward to write an interpreter for the simply typed lambda calculus (STLC) in Haskell. However, how can we be sure that our interpreter is written correctly? Using some features of dependent types – notably, generalized algebraic datatypes, or GADTs – we can incorporate the STLC's type discipline into our interpreter. Using the extra features in Dependent Haskell, we can then write both a big-step semantics and a small-step semantics and have GHC check that they correspond.

3.1.2.1 Type definitions

Our first step is to write a type to represent the types in our lambda-calculus:

```
data Ty = Unit | Ty :⇝ Ty
infixr 0 :⇝
```

I choose *Unit* as our one and only base type, for simplicity. This calculus is clearly not suitable for computation, but it demonstrates the use of GADTs well. The model described here scales up to a more featureful lambda-calculus.[1] The **infixr** declaration declares that the constructor :⇝ is right-associative, as usual.

We are then confronted quickly with the decision of how to encode bound variables. Let's choose de Bruijn indices [12], as these are well-known and conceptually simple. However, instead of using natural numbers to represent our variables, we'll use a custom *Elem* type:

```
data Elem :: [a] → a → ⋆ where
  EZ ::                Elem (x ': xs) x
  ES :: Elem xs x → Elem (y ': xs) x
```

A value of type *Elem xs x* is a proof that *x* is in the list *xs*. This proof naturally takes the form of a natural number, naming the place in *xs* where *x* lives. The first constructor *EZ* is a proof that *x* is the first element in *x* ': *xs*. The second constructor *ES* says that, if we know *x* is an element in *xs*, then it is also an element in *y* ': *xs*.

We can now write our expression type:

```
data Expr :: [Ty] → Ty → ⋆ where
  Var  :: Elem ctx ty                    → Expr ctx ty
```

[1] For example, see my work on glambda at `https://github.com/goldfirere/glambda`.

```
  Lam :: Expr (arg ': ctx) res                          → Expr ctx (arg ':⇝ res)
  App :: Expr ctx (arg ':⇝ res) → Expr ctx arg → Expr ctx res
  TT  ::                                                  Expr ctx 'Unit
```

Like with *Elem list elt*, a value of type *Expr ctx ty* serves two purposes: it records the structure of our expression, *and* it proves a property, namely that the expression is well-typed in context *ctx* with type *ty*. Indeed, with some practice, we can read off the typing rules for the simply typed lambda calculus direct from *Expr*'s definition. In this way, it is impossible to create an ill-typed *Expr* (ignoring the possibility of $\bot$).

3.1.2.2 Big-step evaluator

We now wish to write both small-step and big-step operational semantics for our expressions. First, we'll need a way to denote values in our language:

```
data Val :: Ty → ⋆ where
  LamVal :: Expr '[arg] res → Val (arg ':⇝ res)
  TTVal  ::                   Val 'Unit
```

Our big-step evaluator has a straightforward type:

```
eval :: Expr '[] ty → Val ty
```

This type says that a well-typed, closed expression (that is, the context is empty) can evaluate to a well-typed, closed value, of the same type *ty*. Only a type-preserving evaluator will have that type, so GHC can check the type-soundness of our lambda calculus as it compiles our interpreter.

Of course, to implement *eval*, we'll need several auxiliary functions, each with intriguing types:

```
  -- Shift the de Bruijn indices in an expression
shift :: ∀ ctx ty x. Expr ctx ty → Expr (x ': ctx) ty

  -- Substitute one expression into another
subst :: ∀ ctx s ty. Expr ctx s → Expr (s ': ctx) ty → Expr ctx ty

  -- Perform β-reduction
apply :: Val (arg ':⇝ res) → Expr '[] arg → Expr '[] res
```

The type of *shift* is precisely the content of a weakening lemma: that we can add a type to a context without changing the type of a well-typed expression. The type of *subst* is precisely the content of a substitution lemma: that given an expression of type *s* and an expression of type *t* (typed in a context containing a variable bound to *s*), we can substitute and get a new expression of type *t*. The type of *apply* shows that it does β-reduction: it takes an abstraction of type *arg* ':⇝ *res* and an argument of type *arg*, producing a result of type *res*.

The implementations of these functions, unsurprisingly, read much like the proof of the corresponding lemmas. We even have to "strengthen the induction hypothesis" for *shift* and *subst*; in this context, I mean that we need an internal recursive function with extra arguments. Here are the first few lines of *shift* and *subst*:

```
shift = go []
  where
    go :: ∀ ty. Π ctx₀ → Expr (ctx₀ '++ ctx) ty → Expr (ctx₀ '++ x ': ctx) ty
    go = ...
subst e = go []
  where
    go :: ∀ ty. Π ctx₀ → Expr (ctx₀ '++ s ': ctx) ty → Expr (ctx₀ '++ ctx) ty
    go = ...
```

As many readers will be aware, to prove the weakening and substitution lemmas, it is necessary to consider the possibility that the context change is not happening at the beginning of the list of types, but somewhere in the middle. This generality is needed in the *Lam* case, where we wish to use an induction hypothesis; the typing rule for *Lam* adds the type of the argument to the context, and thus the context change is no longer at the beginning of the context.

Naturally, this issue comes up in our interpreter's implementation, too. The *go* helper functions have types generalized over a possibly non-empty context prefix, ctx_0. This context prefix is appended to the existing context using '++, the promoted form of the existing ++ list-append operator. (Using ' for promoting functions is a natural extension of the existing convention of using ' to promote constructors from terms to types.) The *go* functions also Π-quantify over ctx_0, meaning that the value of this context prefix is available in types (as we can see) and also at runtime. This is necessary because the functions need the length of ctx_0 at runtime, in order to know how to shift or substitute. Note also the syntax Π ctx_0 →, where the Π-bound variable is followed by an →. The use of an arrow here (as opposed to a .) indicates that the parameter is *visible* in source programs; the empty list is passed in visibly in the invocation of *go*. (See also Section 5.1.3.) The final interesting feature of these types is that they re-quantify *ty*. This is necessary because the recursive invocations of the functions may be at a different type than the outer invocation. The other type variables in the types are lexically bound by the ∀ in the type signature of the outer function.

The implementation of these functions is fiddly and uninteresting, and is omitted from this text. However, writing this implementation is made much easier by the precise types. If I were to make a mistake in the delicate de Bruijn shifting operation, I would learn of my mistake immediately, without any testing. In such a delicate operation, this is wonderful, indeed.

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

```
eval (Var v)       = 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*.

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

first check whether or not *e1* can take a step. If it can, we get the result of the step *e1'*, *and* a proof that *'eval e1* $\sim$ *'eval e1'*. This proof enters into the type-checking context and is invisible in the program text. On the right-hand side of the match, we conclude that *App e1 e2* steps to *App e1' e2*. This requires a proof that *'eval* (*App e1 e2*) $\sim$ *'eval* (*App e1' e2*). Reducing *'eval* on both sides of that equality gives us *'eval* (*'apply* (*'eval e1*) *e2*) $\sim$ *'eval* (*'apply* (*'eval e1'*) *e2*). Since we know *'eval e1* $\sim$ *'eval e1'*, however, this equality is easily solvable; GHC does the heavy lifting for us. Similar reasoning proves the equality in the second branch of the **case**, and the other clauses of *step* are straightforward.

The ease in which these equalities are solved is unique to Haskell. I have translated this example to Coq, Agda, and Idris; each has its shortcomings:

- Coq deals quite poorly with indexed types, such as *Expr*. The problem appears to stem from Coq's weak support for dependent pattern matching. For example, if we inspect a *ctx* to discover that it is empty, Coq, by default, forgets the equality *ctx* = [ ]. It then, naturally, fails to use the equality to rewrite the types of the right-hand sides of the pattern match. This can be overcome through various tricks, but it is far from easy. Furthermore, even once these challenges are surmounted, it is necessary to prove that *eval* terminates – a non-trivial task – for Coq to accept the function.

- Agda does a better job with indexed types, but it is not designed around implicit proof search. A key part of Haskell's elegance in this example is that pattern-matching on a *StepResult* reveals an equality proof to the type-checker, and this proof is then used to rewrite types in the body of the pattern match. This all happens without any direction from the programmer. In Agda, on the other hand, the equality proofs must be unpacked and used with Agda's **rewrite** tactic. In Agda, disabling the termination checker for *eval* is easy: simply use the `{-# NO_TERMINATION_CHECK #-}` directive.

- Idris also runs into some trouble with this example. Like Agda, Idris works well with indexed types. The *eval* function is unsurprisingly inferred to be partial, but this is easy enough to fix with a well-placed **assert_total**. However, Idris's proof search mechanism appears to be unable to find proofs that *step* is correct in the *App* cases. (Using an **auto** variable, Idris is able to find the proofs automatically in the other *step* clauses.) Idris comes the closest to Haskell's brevity in this example, but it still requires two explicit, if short, places where equality proofs must be explicitly manipulated.

The above limitations in Coq, Agda, and Idris are from my experimentation. I am not an expert in any of these languages, and I will consult experts before the final dissertation.

3.1.2.4 Conclusion

We have built up a small-step stepper whose behavior is verified against a big-step evaluator. Despite this extra checking, the *step* function will run in an identical manner to one that is unchecked – there is no runtime effect of the extra verification. We can be sure of this because we can audit the types involved and see that only the expression itself is around at runtime; the rest of the arguments (the indices and the equality proofs) are erased. Furthermore, getting this all done is easier and more straightforward in Dependent Haskell than in the other three dependently typed languages I tried. Key to the ease of encoding in Haskell is that Haskell does not worry about termination (more discussion in Section 3.3.3) and has an aggressive rewriting engine used to solve equality predicates.

3.1.3 Units-of-measure

3.1.4 Machine-checked sorting algorithm

3.1.5 Type-safe database access

See also other examples in the work of Oury and Swierstra [37] and Lindley and McBride [32].

3.2 Encoding hard-to-type programs

3.2.1 Variable-arity zipWith

This will be adapted from previous work [17].

3.2.2 Typed reflection

Reflection is the act of reasoning about a programming language from within programs written in that language.[2] In Haskell, we are naturally concerned with reflecting Haskell types. A reflection facility such as the one described here will be immediately applicable in the context of Cloud Haskell. Cloud Haskell [19] is an ongoing project, aiming to support writing a Haskell program that can operate on several machines in parallel, communicating over a network. To achieve this goal, we need a way of communicating data of all types over a wire – in other words, we need dynamic types. On the receiving end, we would like to be able to inspect a dynamically typed datum, figure out its type, and then use it at the encoded type. For more information about how kind equalities fit into Cloud Haskell, please see the GHC wiki at `https://ghc.haskell.org/trac/ghc/wiki/DistributedHaskell`.

[2] Many passages in this example come verbatim from a draft paper of mine [14].

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 $a :\approx: b$ 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 *TyCon*s, returning the heterogeneous equality witness:

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

It is critical that this function returns $(:\approx:)$, not $(:\sim:)$. This is because *TyCon*s exist at many different kinds. For example, *Int* is at kind $\star$, and *Maybe* is at kind $\star \to \star$. 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* :≈: *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[3], 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:

[3] `https://ghc.haskell.org/trac/ghc/wiki/StaticPointers`

```
dynApply (Dyn (TyApp
                 (TyApp (TyCon tarrow) targ)
                 tres)
               fun)
         (Dyn targ' arg)
  | Just HRefl ← eqTyCon tarrow (tyCon :: TyCon (→))
  , Just HRefl ← eqT targ targ'
  = Just (Dyn tres (fun arg))
dynApply _ _ = Nothing
```

We first match against the expected type structure – the first *Dyn* argument must be a function type. We then confirm that the *TyCon tarrow* is indeed the representation for $(\rightarrow)$ (the construct *tyCon*:: *TyCon* $(\rightarrow)$ retrieves the compiler-generated representation for $(\rightarrow)$) and that the actual argument type matches the expected argument type. If everything is good so far, we succeed, applying the function in *fun arg*.

3.2.2.4 Discussion

Heterogeneous equality is necessary throughout this example. It first is necessary in the definition of *eqT*. In the *TyApp* case, we compare *a1* to *a2*. If we had only homogeneous equality, it would be necessary that the types represented by *a1* and *a2* be of the same kind. Yet, we can't know this here! Even if the types represented by *TyApp a1 b1* and *TyApp a2 b2* have the same kind, it is possible that *a1* and *a2* would not. (For example, maybe the type represented by *a1* has kind $\star \rightarrow \star$ and the type represented by *a2* has kind *Bool* $\rightarrow \star$.) With only homogeneous equality, we cannot even write an equality function over this form of type representation. The problem repeats itself in the definition of *dynApply*, when calling *eqTyCon tarrow TArrow*. The call to *eqT* in *dynApply*, on the other hand, *could* be homogeneous, as we would know at that point that the types represented by *targ* and *targ'* are both of kind $\star$.

In today's Haskell, the lack of heterogeneous equality means that *dynApply* must rely critically on *unsafeCoerce*. With heterogeneous equality, we can see that *dynApply* can remain safely outside the trusted code base.

3.2.3 Inferred algebraic effects

This section will contain a Haskell translation of Idris's implementation of algebraic effects [4]. The algebraic effects library allows Idris code to compose effects in a more modular way than can be done with Haskell's monad transformers. It relies critically on dependent types.

3.3 Why Haskell?

There already exist several dependently typed languages. Why do we need another? This section presents several reasons why I believe the work described in this dissertation will have impact.

3.3.1 Increased reach

Haskell currently has some level of adoption in industry.[4] Haskell is also used as the language of choice in several academic programs used to teach functional programming. There is also the ongoing success of the Haskell Symposium. These facts all indicate that the Haskell community is active and sizeable. If GHC, the primary Haskell compiler, offers dependent types, more users will have immediate access to dependent types than ever before.

The existing dependently typed languages were all created, more or less, as playgrounds for dependently typed programming. For a programmer to choose to write her program in an existing dependently typed language, she would have to be thinking about dependent types (or the possibility of dependent types) from the start. However, Haskell is, first and foremost, a general purpose functional programming language. A programmer might start his work in Haskell without even being aware of dependent types, and then as his experience grows, decide to add rich typing to a portion of his program.

With the increased exposure GHC would offer to dependent types, the academic community will gain more insight into dependent types and their practical use in programs meant to get work done.

3.3.2 Backward-compatible type inference

Working in the context of Haskell gives me a stringent, immovable constraint: my work must be backward compatible. In the new version of GHC that supports dependent types, all current programs must continue to compile. In particular, this means that type inference must remain able to infer all the types it does today, including types for definitions with no top-level annotation. Agda and Idris require a top-level type annotation for every function; Coq uses inference where possible for top-level definitions but is sometimes unpredictable. Furthermore, Haskellers expect the type inference engine to work hard on their behalf and should rarely resort to manual proving techniques.

[4] At the time of writing, `https://wiki.haskell.org/Haskell_in_industry` lists 81 companies who use Haskell to some degree. That page, of course, is world-editable and is not authoritative. However, I am personally aware of Haskell's (growing) use in several industrial settings, and I have seen quite a few job postings looking for Haskell programmers in industry. For example, see `http://functionaljobs.com/jobs/search/?q=haskell`.

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 to be 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 $\bot$ 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
```

The type family *Any* can be used at any kind, and so inhabits all kinds.

Furthermore, Dependent Haskell has the $\star :: \star$ axiom, meaning that instead of having an infinite hierarchy of universes characteristic of Coq, Agda, and Idris, Dependent Haskell has just one universe which contains itself. It is well-known that self-containment of this form leads to logical inconsistency by enabling the construction of a looping term [20], but we are unbothered by this. By allowing ourselves to have $\star :: \star$, the type system is much simpler than in systems with a hierarchy of universes.

There are two clear downsides of the lack of totality:

- What appears to be a proof might not be. Suppose we need to prove that type τ equals type σ in order to type-check a program. We can always use $\bot :: \tau :\approx: \sigma$ to prove this equality, and then the program will type-check. The problem will be discovered only at runtime. Another way to see this problem is that equality proofs must be run, having an impact on performance.

  This drawback is indeed serious, and important future work includes designing and implementing a totality checker for Haskell. (See the work of Vazou et al. [47] for one approach toward this goal. Recent work by Karachalias et al. [29] is another key building block.) Unlike in other languages, though, the totality checker would be chiefly used in order to optimize away proofs, rather than to keep the language safe. Once the checker were working, we could also add compiler flags to give programmers compile-time warnings or errors about partial functions, if requested.

- Lack of termination in functions used at the type level might conceivably cause GHC to loop. This is not a great concern, however, because the loop is directly caused by a user's type-level program. In practice, GHC counts steps it uses in reducing types and reports an error after too many steps are taken. The user can, via a compiler flag, increase the limit or disable the check.

The advantages to the lack of totality checking are that Dependent Haskell is simpler for not worrying about totality, and also that by using Dependent Haskell, we will learn more about dependent types in the presence of partiality.

3.3.4 GHC is an industrial-strength compiler

Hosting dependent types within GHC is likely to reveal new insights about dependent types due to all of the features that GHC offers. Not only are there many surface language extensions that must be made to work with dependent types, but the back end must also be adapted. A dependently typed intermediate language must, for example, allow for optimizations. Working in the context of an industrial-strength compiler also forces the implementation to be more than just "research quality", but ready for a broad audience.

3.3.5 Manifest type erasure properties

A critical property of Haskell is that it can erase types. Despite all the machinery available in Haskell's type system, all type information can be dropped during compilation. In Dependent Haskell, this does not change. However, dependent types certainly blur the line between term and type, and so what, precisely, gets erased can be less clear. Dependent Haskell, in a way different from other dependently typed languages, makes clear which arguments to functions (and data constructors) get erased. This is through the user's choice of relevant vs. irrelevant quantifiers, as explored in Section 5.1.2. Because erasure properties are manifestly available in types, a performance-conscious user can audit a Dependent Haskell program and see exactly what will be removed at runtime, aiming to meet any particular performance goals the user has.

It is possible that, with practice, this ability will become burdensome, in that the user has to figure out what to keep and what to discard. Idris's progress toward type erasure analysis [6, 46] may benefit Dependent Haskell as well.

3.3.6 Haskellers want dependent types

The design of Haskell has slowly been marching toward having dependent types. Haskellers have enthusiastically taken advantage of the new features. For example, over 1,000 packages published at `hackage.haskell.org` use type families [43]. Anecdotally, Haskellers are excited about getting full dependent types, instead of just faking them [16, 33, 34]. Furthermore, with all of the type-level programming features that exist in Haskell today, it is a reasonable step to go to full dependency.

Chapter 4

System FC

This chapter will be a thorough review of System FC as it exists today. It will be a combination of the systems described in previous work [7, 18, 53]. My chief goal in writing this chapter is establishing a solid baseline from which to grow. System FC has gotten big enough that 12-page papers do not have enough space to fully explain it; all recent treatments have been abbreviated. This treatment will be unabridged, but it will focus on the theoretical version of FC, as distinct from the variant that is implemented in GHC.

4.1 Kind polymorphism

4.2 Roles

4.3 Coercions

4.4 Axioms

4.5 Contexts

4.6 Type safety proof sketch

Instead of proving the type safety of System FC, I will draw a very broad outline. This outline will be filled in with the proof of System FCD, the version with dependent types.

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 *quantifiee* 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 quantifiee may be used in the quantifier body; a non-dependent quantifiee may not.

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

$$id :: \forall\ a.\ a \to a$$

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

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

$$pred :: Int \to Int$$

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

Dependent Haskell adds a new dependent quantifier, Π, as discussed below.

A key requirement of dependent arguments – that is, concrete choices of instantiations of dependent quantifiees – is that they are expressible in types. See Section 5.4 for a discussion of how this plays out in practice.

5.1.2 Relevance

A quantifier may be either relevant or irrelevant. A relevant quantifiee may be used in a relevant position in the *function* quantified over; an irrelevant quantifiee may be used only in irrelevant positions. Note that relevance talks about usage in the function quantified over, not the type quantified over (which is covered by the *dependency* property).

Relevance is very closely tied to type erasure. Relevant arguments in terms are precisely those arguments that are not erased. However, the *relevance* property applies equally to type-level functions, where erasure does not make sense, as all types are erased. For gaining an intuition about relevance, thinking about type erasure is a very good guide.

Today's Haskell uses $(\rightarrow)$ for relevant quantification. For example, here is the body of *pred*:

$$pred\ x = x - 1$$

Note that x, a relevant quantifiee, is used in a relevant position on the right-hand side. Relevant positions include all places in a term or type that are not within a type annotation or other type-level context, as will be demonstrated in the next example.

Today's Haskell uses $\forall$ for irrelevant quantification. For example, here is the body of *id*:

$$id\ x = (x :: a)$$

The type variable a is the irrelevant quantifiee. According to Haskell's scoped type variables, it is brought into scope by the $\forall\ a$ in *id*'s type annotation. (It could also be brought into scope by using a in a type annotation on the pattern x to the left of the =.) Although a is used in the body of *id*, it is used only in an irrelevant position, in the type annotation for x. It would violate the irrelevance of $\forall$ for a to be used outside of a type annotation or other irrelevant context. As functions can take irrelevant arguments, irrelevant contexts include these irrelevant arguments.

Dependent Haskell adds a new relevant quantifier, Π. The fact that Π is both relevant and dependent is the very reason for Π's existence!

5.1.3 Visibility

A quantifier may be either visible or invisible. The argument used to instantiate a visible quantifiee appears in the Haskell source; the argument used to instantiate an invisible quantifiee is elided. Some readers may prefer the terms *explicit* and *implicit*

Quantifier	Dependency	Relevance	Visibility
$\forall\ (a :: \tau).\ ...$	dependent	irrelevant	invisible (unification)
$\forall\ (a :: \tau) \to ...$	dependent	irrelevant	visible
$\Pi\ (a :: \tau).\ ...$	dependent	relevant	invisible (unification)
$\Pi\ (a :: \tau) \to ...$	dependent	relevant	visible
$\tau \Rightarrow ...$	non-dependent	relevant	invisible (solving)
$\tau \to ...$	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 $(\to)$ 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*. *Show a* $\Rightarrow$ *a* $\to$ *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 $(\to)$-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 Π; 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

first one ($\forall\ (a :: t).\ ...$) and the last two ($\Rightarrow$ and $\rightarrow$) exist in today's Haskell and are completely unchanged. Dependent Haskell adds a visible $\forall$ and the two Π quantifiers.[1]

The visible $\forall$ is useful in situations where a type parameter might otherwise be ambiguous. For example, suppose F is a type family and consider this:

$$frob :: \forall\ a.\ F\ a \rightarrow F\ [a]$$

This type signature is inherently ambiguous, and GHC reports an error when it is written. Suppose that we know we want a particular use of *frob* to have type $Int \rightarrow Bool$. Even with that knowledge, there is no way to determine how to instantiate a. To fix this problem, we simply make a visible:

$$frob :: \forall\ a \rightarrow F\ a \rightarrow F\ [a]$$

Now, any call to *frob* must specify the choice for a, and the type is no longer ambiguous.

A Π-quantified parameter is both dependent (it can be used in types) and relevant (it can be used in terms). Critically, pattern-matching (in a term) on a Π-quantified parameter informs our knowledge about that parameter as it is used in types, a subject we explore next.

5.2 Pattern matching

This section will address how dependent pattern matching works, illustrated by examples. The key points will be

- A dependent pattern match is one where the term-level match also informs a Π-quantified variable in types.
- Dependent Haskell source will include only one **case** construct. Whether or not to do dependent matching will depend on the context. Specifically, a pattern match will be dependent iff:
  1. The scrutinee (the expression between **case** and **of**) mentions only Π-quantified variables and expressions in the shared subset (Section 5.4), *and*
  2. The result type of the pattern-match is known (via bidirectional type inference).

  I will also explain why we need these conditions.

[1] The choice of syntax here is directly due to the work of Gundry [21].

5.3 Inferring Π

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

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 6

System FCD

This chapter will build on Chapter 4 by adding kind equalities and dependent types. It will be a combination of the systems in Weirich et al. [52] and Gundry [21].

6.1 Merging types and kinds

6.2 Levity polymorphism

Levity polymorphism is described in a recent draft paper [14].

6.3 Roles and dependent types

In contrast to Section 5.5, which will describe the user-facing interaction between roles and dependent types, this section will describe the internals.

6.4 Metatheory

This section will sketch the necessary lemmas needed to prove type safety, along with the so-called Lifting Lemma, which demonstrates an important property of the coercion language. The actual proofs will appear in Appendix B.

Chapter 7

A specification of Dependent Haskell

This section will lay out a formal static semantics for Haskell, including **data** and **type family** definitions. This semantics will then be used in Chapter 8 as the specification for the type inference algorithm.

7.1 Bidirectional type systems

Critical to Dependent Haskell is its reliance on a bidirectional type system [41]. The bidirectional system is used to allow higher-rank polymorphism [40] as well as dependent pattern matching (Section 5.2). The presentation here will be along the lines of that of Dunfield and Krishnaswami [13].

7.2 Types

7.3 Terms

7.4 Type declarations

Chapter 8

Type inference with dependent types

This chapter will lay out the type inference algorithm, including how it elaborates a Dependent Haskell program into System FCD. Soundness and completeness properties will be considered, though the bulk of the proofs will be deferred to Appendix C. This work will build directly on that of Vytiniotis et al. [49].

8.1 Type inference algorithm

8.2 Soundness properties

8.3 Completeness properties

8.3.1 On incompleteness in type inference

Chapter 9

Implementation notes

This chapter will consider any differences between the theory presented in this dissertation and the actual implementation in GHC. It is possible that nothing of general interest will fit in this chapter and that it might be removed.

One possible subject of interest in this section is that the design of Dependent Haskell appears to require having just one parser for terms, types, and kinds. This is in stark contrast to today's GHC, which has three separate parsers.

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 Π, 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

can these be improved? More generally, how can error messages be customized by programmers to fit their domain?

- What editor support is necessary to make dependent types in Haskell practical?
- Are there constructs that I have been unable to promote? How can these be made to work in types?
- How do we optimize a dependently typed program? Ideally, a program should be optimized to the same level whether an argument is dependent or not. However, optimizing Π-quantified arguments will amount to proving the optimizations correct! How do we retain runtime performance in the face of dependent types?
- How will dependent types interact with type-checker plugins [22]? Can we use an SMT solver to make working with dependent types easier?
- Dependent types will allow for proper dependent pairs (Σ-types). Is it worth introducing new syntax to support these useful constructs directly?

Chapter 11

Timeline

Much of the work described in this proposal has already taken place:

1. The entirety of Chapter 4 is simply collecting together previous work on System FC.
2. Although some details have yet to be worked out (such as import/export lists and the concrete syntax around promoting functions), much of the design of Dependent Haskell has been worked out, as well.
3. The system described in previous work [52] will form the basis of System FCD.
4. I have an implementation[1] available of GHC with merged types and kinds and full kind equalities. This implementation successfully compiles itself and all of GHC's standard libraries, as well as some examples testing the power of kind equalities.

Here are the major remaining tasks to complete:

1. System FCD will need to include dependent types This addition will be strongly influenced by the work of Gundry [21]. There is some work to be done here, for sure, but no more than a solid month's worth, I estimate.
2. The surface language specification and type inference algorithm are already in draft form (though it is not in a state ready to share), with an attempt at a proof started. This area is surely the murkiest area of this dissertation. However, given the previous work in this area [21], I am confident that this will work out. I hope to finish the bulk of this work in a 2-month time frame.
3. I need to finish polishing my existing implementation of kind equalities and merge with GHC's master branch. This will take 2 months, I estimate.
4. I need to implement Π into GHC. I estimate this will take 3 months.

[1] `https://github.com/goldfirere/ghc/`, the `nokinds` branch

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 Π 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.

Appendix A

Typographical conventions

This dissertation is typeset using LaTeX with considerable help from lhs2TeX[1] and ott [42]. The lhs2TeX software allows Haskell code to be rendered more stylistically than a simple `verbatim` environment would allow. The table below maps Haskell source to glyphs appearing in this dissertation:

Haskell	Typeset	Description
`->`	$\rightarrow$	function arrow and other arrows
`=>`	$\Rightarrow$	constraint arrow
`*`	$\star$	the kind of types
`forall`	$\forall$	dependent irrelevant quantifier
`pi`	Π	dependent relevant quantifier
`++`	$+\!\!+$	list concatenation
`:~~:`	$:\approx:$	heterogeneous propositional equality
`:~>`	$:\rightsquigarrow$	lambda-calculus arrow (from Section 3.1.2)
`undefined`	$\bot$	canonical looping term

Table A.1: Typesetting of Haskell constructs

In addition to the special formatting above, I assume a liberal overloading of number literals, including in types. For example, I write 2 where I really mean *Succ* (*Succ Zero*), depending on the context.

[1] `http://www.andres-loeh.de/lhs2tex/`

Appendix B

Proof of type safety

Appendix C

Proofs about type inference

Bibliography

[1] Lennart Augustsson. Cayenne—a language with dependent types. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250. ACM, 1998.

[2] Christiaan P. R. Baaij. *Digital Circuits in CλAsh: Functional Specification and Type-Directed Synthesis*. PhD thesis, University of Twente, 2015.

[3] Patrick Bahr. Composing and decomposing data types: A closed type families implementation of data types à la carte. In *Workshop on Generic Programming*. ACM, 2014.

[4] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *International Conference on Functional Programming*. ACM, 2013.

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

[6] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*. 2004.

[7] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for Haskell. In *International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[8] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyon Jones. Associated type synonyms. In *International Conference on Functional Programming*, ICFP '05. ACM, 2005.

[9] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[10] Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

[11] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

[12] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381392, 1972. ISSN 1385-7258. doi: 10.1016/1385-7258(72)90034-0. URL `http://dx.doi.org/10.1016/1385-7258(72)90034-0`.

[13] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

[14] Richard A. Eisenberg. An overabundance of equality: Implementing kind equalities into haskell. Draft, 2015.

[15] Richard A. Eisenberg and Jan Stolarek. Promoting functions to type families in Haskell. In *Symposium on Haskell*, Haskell '14. ACM, 2014.

[16] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

[17] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations (extended version). Technical Report MS-CIS-13-10, University of Pennsylvania, 2013.

[18] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*, POPL '14. ACM, 2014.

[19] Jeff Epstein, Andrew P. Black, and Simon Peyton Jones. Towards haskell in the cloud. In *Haskell Symposium*. ACM, 2011.

[20] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[21] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[22] Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in ghc haskell. To appear at Haskell Symposium 2015, 2015. URL `http://adam.gundry.co.uk/pub/typechecker-plugins/typechecker-plugins-2015-03-03.pdf`.

[23] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2), March 1996.

[24] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Workshop on Haskell*. ACM, 2003.

[25] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, 1969.

[26] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Conference on History of Programming Languages*, 2007.

[27] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3), May 2000.

[28] Mark P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, 2000.

[29] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. GADTs meet their match. In *International Conference on Functional Programming*, ICFP '15. ACM, 2015.

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

[31] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Workshop on Types in Languages Design and Implementation*. ACM, 2003.

[32] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *ACM SIGPLAN Haskell Symposium*, 2013.

[33] Conor McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

[34] Conor McBride. The Strathclyde Haskell Enhancement. `https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`, 2011.

[35] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17, 1978.

[36] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[37] Nicolas Oury and Wouter Swierstra. The power of Pi. In *Proc. 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 39–50. ACM, 2008.

[38] Conrad Parker. Type-level instant insanity. *The Monad.Reader*, (8), 2007.

[39] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[40] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1), January 2007.

[41] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1), January 2000.

[42] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1), January 2010.

[43] Jan Stolarek, Simon Peyon Jones, and Richard A. Eisenberg. Injective type families for Haskell. Draft, 2015.

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

[45] Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1), January 2007.

[46] Matúš Tejiščák and Edwin Brady. Practical erasure in dependently typed languages. Draft, 2015. URL `http://eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf`.

[47] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for Haskell. In *International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[48] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let should not be generalized. In *Types in Language Design and Implementation*, TLDI '10. ACM, 2010.

[49] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X) modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5), September 2011.

[50] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76. ACM, 1989.

[51] Geoffrey Washburn and Stephanie Weirich. Boxes Go Bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *International Conference on Functional Programming*. ACM, 2003.

[52] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

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