Type variables in patterns

Richard A. Eisenberg
Bryn Mawr College
Bryn Mawr, PA, USA
rae@cs.brynmawr.edu

Joachim Breitner
University of Pennsylvania
Philadelphia, PA, USA
joachim@cis.upenn.edu

Simon Peyton Jones
Microsoft Research
Cambridge, UK
simonpj@microsoft.com

Abstract
For many years, GHC has implemented an extension to Haskell that allows type variables to be bound in type signatures and patterns, and to scope over terms. This extension was never properly specified. We rectify that oversight here. With the formal specification in hand, the otherwise-labyrinthine path toward a design for binding type variables in patterns becomes blindingly clear. We thus extend ScopedTypeVariables to bind type variables explicitly, obviating the Proxy workaround to the dustbin of history.

1 Introduction
Haskell allows the programmer to write a type signature for a definition or expression, both as machine-checked documentation, and to resolve ambiguity (Section 2.1). For example,

\[
\text{prefix} :: a \rightarrow [[a]] \rightarrow [[a]]
\]

\[
\text{prefix} \ x \ yss = \text{map} \ xcons \ yss
\]

\[
\text{where} \ xcons \ ys = x : ys
\]

Sadly, it is not always possible to write such a type signature. For example, to give a signature for \textit{xcons} we might try:

\[
\text{prefix} :: a \rightarrow [[a]] \rightarrow [[a]]
\]

\[
\text{prefix} \ x \ yss = \text{map} \ xcons \ yss
\]

\[
\text{where} \ xcons :: [a] \rightarrow [a]
\]

\[
xcons \ ys = x : ys
\]

But Haskell98’s scoping rules specify that the \(a\) in the signature for \textit{xcons} is locally quantified, thus: \(xcons :: \forall a. [a] \rightarrow [a]\). That is not what we want! We want the \(a\) in the signature for \textit{xcons} to mean “the universally quantified type variable for \textit{prefix}”, and Haskell98 provides no way to do that.

The inability to supply a type signature for \textit{xcons} might seem merely inconvenient, but it is just the tip of an iceberg. Haskell uses type inference to infer types, and that is a wonderful thing. However, insisting on complete type inference—that is, the ability to infer types for any well-typed program with no help from the programmer—places serious limits on the expressiveness of the type system. GHC’s version of Haskell has gone far beyond these limits, and fundamentally relies on programmer-supplied annotations to guide type inference. As some examples, see the work of Peyton Jones et al. [2007], Vytiniotis et al. [2011], or Eisenberg et al. [2016].

So the challenge we address is this: it should be possible for the programmer to write an explicit type signature for any sub-term of the program. To do so, some type signatures must refer to a type that is already in the static environment, so we need a way to name such types. The obvious way to address this challenge is by providing language support for lexically scoped type variables. GHC has long supported scoped type variables: the \textit{ScopedTypeVariables} extension is very popular, and 29% of Haskell packages on Hackage use it. But it has never been formally specified! Moreover, as we shall see, it is in any case inadequate to the task. In this paper we fix both problems, making the following contributions:

- In the days of Haskell98, scoped type variables were seldom crucial. Through a series of examples we show that, as Haskell’s type system has grown more sophisticated, the need for scoped type variables has become acute (Section 2), while at the same time GHC’s existing support for them has become more visibly inadequate (Section 3).
- To fix these inadequacies, we describe visible type application in patterns, a natural extension to GHC’s existing visible type applications from terms to patterns (Section 4).
- We give the first formal specification of scoped type variables for Haskell, formalizing the folklore, and providing a firm foundation for both design and implementation (Section 5).
- As part of this specification, we offer a new and simpler typing judgment for GADT pattern matching (Section 5.3), which treats uniformly the universal and existential variables of a data constructor.

2 Motivation and background
2.1 The need for type annotations
One of the magical properties of ML-family languages, including Haskell, is that type inference allows us to write many programs with no type annotations whatsoever. In practice, however, Haskell programs contain many user-written type signatures, for two main reasons.

First, the type of a function can be extremely helpful as documentation, with the advantage that it is machine-checked documentation. Almost all programmers regard it as good practice to provide a signature for every top-level function. Indeed, GHC has a warning flag, -Wmissing-signatures, which enforces this convention.
Figure 1. Types of standard functions

Second, as Haskell’s type system becomes increasingly expressive, complete type inference becomes intractable, and the type system necessarily relies on programmer-supplied type annotations. Here are some examples:

- **Type-class ambiguity** is present even in Haskell 98. Consider¹:

  ```
  normalize :: String → String
  normalize s = show (read s)
  ``

  This function parses a string to a value of some type, and then turns that value back into a string. But nothing in the code specifies that type, so the programmer must disambiguate. One way to do so is to provide a type signature that specifies the result type of the `read s` call, thus:

  ```
  normalize s = show (read s :: Int)
  ``

- **Polymorphic recursion**. In ML, recursive calls to a function must be at monomorphic type, but Haskell has always supported polymorphic recursion, provided the function has a type signature. For example:

  ```
  data T a = Leaf a | Node (T [a]) (T [a])
  leaves :: T a → [a]
  leaves (Leaf x) = [x]
  leaves (Node t1 t2) = concat (leaves t1 ++ leaves t2)
  ``

- **Higher-rank types** [Peyton Jones et al. 2007]. Consider

  ```
  f :: ∀a. [a] → [a]
  g = (g "He110", g [True, False])
  ``

  Here the type of `g` is polymorphic, so it can be applied to lists of different type. The type signature is essential to specify the type of the argument `g`; without it, `f` will be rejected.

- **Generalized algebraic data types** [Schrijvers et al. 2009]. The popular GADTs extension to GHC allows pattern matching to refine the type information available in the right hand side of an equation. Here is an example:

  ```
  data G a where
  MkInt :: G Int
  MkFun :: G (Int → Int)
  ``

  When we learn that a value `g :: G a` is actually the constructor `MkInt`, then we simultaneously learn that `a` really is `Int`. GHC can use this fact during type checking the right-hand side of a function, like this:

  ```
  matchG :: G a → a
  matchG MkInt = 5
  matchG MkFun = (+10)
  ``

  Again, however, `matchG` will only type-check if it is given a signature; see Schrijvers et al. [2009] for details.

  - **Ambiguous types.** Consider

    ```
    type family F a
    ambig :: Typeable a ⇒ F a → Int
    test :: Char → Int
    test x = ambig x
    ``

    In `test` GHC must decide at what type to call `ambig`; that is, what type should instantiate the `a` in `ambig`’s type. Any choice `a = τ` must ensure that `F τ ∼ Char` but, because `F` might not be injective, that does not tell us what `a` should be. A type signature is not enough to resolve this case; we need a different form of type annotation, namely `visible type application` (Section 2.3).

    There is a general pattern here: as the type system becomes more expressive, the type inference needs more guidance. Moreover, that guidance is extremely informative to the programmer, as well as to the compiler.

2.2 Support for scoped type variables

Given the increasing importance of type annotations, a good principle is that it should be possible to specify, via a type signature, the type of any sub-expression or any let-binding. Alas, as shown in the introduction, Haskell98 supports only closed type signatures, so there are useful type signatures that we simply cannot write.

The key deficiency in Haskell98 is that it provides no way to bring type variables into scope. GHC has recognized this need for many years, and the `ScopedTypeVariables` extension offers two ways to bring a type variable into scope:

- **Binding in a declaration signature.** Since 2004 GHC allows you to write

  ```
  prefix :: ∀a. [a] → [[a]]
  prefix x yss = map xcons yss
  where xcons :: [a] → [a]
  xcons ys = x : ys
  ``

  The explicit “∀” brings `a` into scope in the rest of the type signature (of course), but it also brings `a` into scope in the body of the named function, `prefix` in this case. The rule is a bit strange, because the definition of `prefix` is not syntactically “under” the ∀, and indeed the signature can be written far away from the actual binding. But in practice the rule works well, and we take it as-is for the purposes of this paper.

¹ Figure 1 gives the types of standard functions, such as `read` and `show`.
We see above that `ScopedTypeVariables` enables the user to bind type variables in patterns, by providing a pattern signature, that is, a type signature in a pattern. We explore pattern signatures and their shortcomings, in this section.

3 Pattern signatures and their shortcomings

We see above that `ScopedTypeVariables` enables the user to bind type variables in patterns, by providing a pattern signature, that is, a type signature in a pattern. We explore pattern signatures and their shortcomings, in this section.

- **Pattern signatures**: binding a type variable in a pattern. For even longer, since 1998, GHC has allowed you to write this:

  ```haskell
  prefix (x :: b) yss = map xcons yss
  where xcons :: [b] → [b]
       xcons yss = x : yss
  ```

  Here, we bind the type variable `b` in the pattern `(x :: b)`, and this binding scopes over the body of the binding. We describe pattern signatures in much more detail in Section 3.

2.3 Visible type application

The `TypeApplications` extension provides a relatively new form of type annotation: explicit type applications, first described by Eisenberg et al. [2016]. The idea is that an argument of the form `@ty` specifies a `type` argument. This can often be used more elegantly than a type signature. For example, a hypothetical unit-test for the function `isJust` can equivalently be written more elegantly using explicit type annotations:

```haskell
testIsJust1 = isJust (Just 2018 :: Int) == True
testIsJust2 = isJust (Nothing :: Maybe Int) == False
```

Visible type application solves the awkward case of ambiguous in Section 2.1: we can disambiguate the call with a type argument. For example:

```haskell
type family F a
type instance F Bool = Char
ambig :: Typeable a ⇒ F a → Int

test = Char → Int

x = ambig @Bool x
```

Here we specify that `ambig` should be called at `Bool`, and that is enough to type-check the program.

It is natural to wonder whether we can extend visible type application to patterns, just as we extended type signatures to patterns. Doing so is the main language extension suggested in this paper: Section 4.

3.1 The binding structure of a pattern signature

A pattern signature may bind a type variable, but it may also mention a type variable that is already in scope. For example, we may write:

```haskell
prefix (x :: a) yss = map xcons yss
  where xcons (ys :: [a]) = x : ys
```

Here, the pattern signature `(x :: a)` binds `a` (as well as `x`), but the pattern signature `(ys :: [a])` simply mentions `a` (which is already in scope), as well as binding `ys`. The rule is this: a use of a type variable `p` in a pattern signature is an occurrence of `p` if `p` is already in scope; but binds `p` if `p` is not already in scope.

It is entirely possible to have many different type variables in scope, all of which are aliases for the same type. For example:

```haskell
prefix :: ∀a. a → [[a]] → [[a]]

prefix (x :: b) (ys :: [[c]]) = map xcons yss
  where xcons (ys :: [d]) = x : ys
```

Here `a`, `b`, `c`, and `d` are all in scope in the body of `xcons`, and are all aliases for the same type.

The current implementation of `ScopedTypeVariables` allows such lexically-scoped type variables to stand only for other type variables, and not for arbitrary types, a point we return to in Section 3.5.

3.2 Pattern signatures are useful

Pattern signatures have merit even if there are no type variables around. Consider this Haskell program:

```haskell
main = do x ← readLn
        if null x then putStrLn "Empty"
        else putStrLn "Not empty"
```

The types of the program are ambiguous: Clearly, `x` is some type that has a `Read` instance, and because it is passed to `null`, it is a list\(^2\), but the compiler needs to know the precise type, and rejects the program.

To fix this in Haskell98, the programmer has two options:

- They can wrap the call to `readLn` in a type annotation:

  ```haskell
  main = do x ← (readLn :: IO [Int])
  if null x then putStrLn "Empty!"
  else putStrLn x
  ```

  but this is infelicitous because there is no question that `readLn` is in the `IO`, and with larger types this can get very verbose.

- They can wrap an occurrence of `x` in a type annotation:

  ```haskell
  main = do x ← readLn
  if null (x :: [Int]) then putStrLn "Empty"
  else putStrLn x
  ```

\(^2\)Or, with a recent version of the standard library, it is something with a `Foldable` instance.
but again this is unsatisfying, because it feels too late. Both variants are essentially work-arounds for the natural way of specifying the type of \( x \), namely at its binding site:

\[
\begin{align*}
\text{main} &= \text{do} \,(x :: [\text{Int}]) \leftarrow \text{readLn} \\
& \quad \text{if} \, \text{null} \, x \, \text{then} \, \text{putStrLn} \, \text{"Empty"} \\
& \quad \text{else} \, \text{print} \, x
\end{align*}
\]

which is precisely what the \texttt{PatternSignatures} language extension provides\(^3\)—the ability to write a type annotation in a pattern.

Users, especially beginners, who have to track down a confusing type error in their code, can now exhaustively type annotate not just their terms, but also their patterns, until they have cornered the bug.

### 3.3 Pattern signatures are essential

Pattern signatures become more crucial when we consider existential types. The \texttt{ExistentialQuantification} extension allows users to bind \texttt{existential} variables in their data constructors. These are type variables whose values are “stored” by a constructor (but not really, because types are erased) and made available during pattern matching. Here are two examples:

```haskell
data Ticker where
  MkTicker :: \( a \) \to (a \to a) \to (a \to \text{Int}) \to Ticker

data Showable where
  MkShowable :: \( a \) \Rightarrow Show \to a \to Showable
```

A \texttt{Ticker} contains an object of some type (but we do not know \texttt{what} type), along with an update function of type \( a \to a \) and a way to convert an \( a \) into an \texttt{Int}. The \texttt{Showable} type packs a value of an arbitrary type that has a \texttt{Show} instance along with its \texttt{Show} dictionary. Here are some functions that operate on these types:

- Updates a ticker, returning whether or not the ticker
- \texttt{has reached a limit}
- \texttt{tick} :: \texttt{Ticker} \to \texttt{Int} \to \texttt{(Ticker, Bool)}
- \texttt{tick} (\texttt{MkTicker val upd toInt}) limit = \texttt{(newTicker, toInt newVal ⩾ limit)}
- \texttt{where} \texttt{newVal} = \texttt{upd val}
  - \texttt{newTicker} = \texttt{MkTicker newVal upd toInt}

\texttt{showAll} :: \exists a. \texttt{Showable} \Rightarrow \texttt{String}

\texttt{showAll} [] = \texttt{""}

\texttt{showAll} (\texttt{MkShowable x : ss}) = \texttt{show x} \+ \texttt{showAll ss}

We see that the \texttt{tick} function can unpack the existential in the \texttt{Ticker} value and operate on the value of type \( a \) without ever knowing what \( a \) is. Similarly, the \texttt{showAll} function works with data of type \( a \) knowing only that \( a \) has a \texttt{Show} instance.

\(^3\)Modern GHC actually folds \texttt{PatternSignatures} into \texttt{ScopedTypeVariables}, giving both extensions the same meaning. However, it is expositionally cleaner to separate the two, as we do throughout this paper.

The \texttt{Show} \( a \) constraint is brought into scope by the pattern match.

However, existentials can never escape, forbidding the following function:

\[
\text{jailbreak} \,(\texttt{MkTicker val _ _}) = \texttt{val}
\]

What should the type of \texttt{jailbreak} be? There is no answer to this question (\texttt{jailbreak} :: \texttt{Ticker} \to \texttt{a} is clearly too polymorphic), and so GHC rejects this definition, correctly stating that type variable \( \texttt{a} \) would escape its scope.

We naturally wish to name the existential type variable sometimes. For example, suppose we wanted to give \texttt{newVal} (in the \texttt{where} clause of \texttt{tick}) a type signature. Saying \texttt{newVal} :: \texttt{a} would hardly do, because there is not yet a connection between the name \( a \) and the type unpacked from \texttt{MkTicker}. We have to do this:

\[
\text{tick} \,(\texttt{MkTicker} \,(\texttt{val :: b}) \,\texttt{upd toInt}) \texttt{limit} = \ldots
\]

\texttt{where} \texttt{newVal} :: \texttt{b}

\[
\texttt{newVal} = \texttt{upd val}
\]

The \texttt{val :: b} in the pattern binds the type variable \( b \), so we can refer back to it later.

### 3.4 Pattern signatures are clumsy

Pattern signatures can be clumsy to use, when the type variable is buried deep inside an ornate type. Here is a contrived example:

```haskell
data Elab where
  MkElab :: \exists a. Show \Rightarrow \texttt{a} \Rightarrow \texttt{Elab}

getE :: Elab \to \texttt{Int}

getE (\texttt{MkElab} \,(\texttt{xs :: [Maybe (Tree \,(\texttt{a}, \texttt{Int}) \to \texttt{Elab})]}) \)) = \ldots \texttt{a} \ldots
```

To bring \( a \) into scope in \texttt{f}’s \texttt{right-hand side}, we have to repeat the \texttt{MkElab}’s elaborate argument type.

More seriously, it may be impossible, rather than merely clumsy, to bind the variable we need. Consider the following GADT:

```haskell
data GM a where
  MkMaybe :: \exists b. \texttt{GM (Maybe b)}

matchGM :: \exists a. \texttt{GM a} \to \texttt{Bool}

matchGM x (\texttt{MkMaybe \,(\texttt{y :: GM (Maybe b)})}) = \texttt{isJust @b x}
```

This definition works just fine: GHC learns that \( x \)’s type is \( \texttt{Maybe b} \) (for some existential \( b \)) and the call to \texttt{isJust} is well typed. But what if we want to bind \( b \) in this definition? Annoyingly, \texttt{MkMaybe} has no argument to which we can apply a pattern signature. Nor does it work to wrap a pattern signature around the \texttt{outside} of the match, thus:

```haskell
matchGM :: \exists a. \texttt{GM a} \to \texttt{Bool}

matchGM x (\texttt{MkMaybe \,(\texttt{y :: GM (Maybe b)}) \to \texttt{isJust @b x}})
```

This definition is rejected. The problem is that the type annotation on the \texttt{MkMaybe} pattern is checked \texttt{before} the pattern
Type variables in patterns

...itself is matched against\(^4\). Before matching the MkMaybe, we do not yet know that \(a\) is really Maybe \(b\). Nor can we put the type annotation on \(x\), as that, too, occurs before the MkMaybe pattern has been matched. A possible solution is this monstrosity:

\[
\text{matchGM} \colon a \rightarrow \text{GM} a \rightarrow \text{Bool}
\]

\[
\text{matchGM} x \text{gm} @\text{MkMaybe} = \text{case gm of}
\]

\[
(x :: \text{GM} \text{(Maybe } b)) \rightarrow \text{isJust } @b x
\]

This is grotesque. We must do better, and we will in Section 4.

3.5 Pattern signatures resist refactoring

In Section 3.1, we explained that a scoped type variable may refer only to another type variable. This means that the definition

\[
\text{prefix} :: a \rightarrow [\text{a}]) \rightarrow [\text{a}])
\]

\[
\text{prefix} \ (x :: b) \ ys = \text{map xcons} \ ys
\]

where \(x\text{cons} \ ys = x : ys\)

is accepted, because \(b\) stands for the type variable in the type of \(\text{prefix}\). But suppose, for example, we specialize the type signature of \(\text{prefix}\) without changing its definition:

\[
\text{prefix} :: \text{Int} \rightarrow [\text{Int}]) \rightarrow [\text{Int}])
\]

\[
\text{prefix} \ (x :: b) \ ys = \text{map xcons} \ ys
\]

where \(x\text{cons} \ ys = x : ys\)

Now this definition is rejected with the error message "Couldn’t match expected type \(b\) with actual type \(\text{Int}\)". Since \(b\) would have to stand for \(\text{Int}\).

Since the design of ScopedTypeVariables, GHC has evolved, and with the advent of type equalities, the restriction itself becomes confusing. Should GHC accept the following definition?

\[
\text{prefix} :: a \sim \text{Int} \Rightarrow \text{Int} \rightarrow [\text{a}]) \rightarrow [\text{a}])
\]

\[
\text{prefix} \ (x :: b) \ ys = \text{map xcons} \ ys
\]

where \(x\text{cons} \ ys = x : ys\)

Is \(b\) an alias for \(a\) (legal) or for \(\text{Int}\) (illegal)? Since \(a\) and \(\text{Int}\) are equal, the question does not really make sense. We therefore propose to simply drop this restriction (Section 4.3).

3.6 Pattern signatures are inadequate

We end our growing list of infelicities with a case in which there is no way whatsoever to bind the type variable, short of changing the data type definition.

Type families [Chakravarty et al. 2005; Eisenberg et al. 2014] allow users to write type-level functions and encode type-level computation. For example, we might write this:

\[
\text{type family } F \ a \ where
\]

\[
F \ \text{Int} = \text{Bool}
\]

\[
F \ \text{Char} = \text{Double}
\]

\[
F \ \text{Float} = \text{Double}
\]

Naturally, we can use a type family to define the type of an argument to an existential data constructor:

\[
\text{data } T \ where
\]

\[
\text{MkTF} :: \forall a. \text{Typeable } a \Rightarrow F \ a \rightarrow T
\]

The \(\text{MkTF}\) constructor stores a value of type \((F \ a)\); it also stores a dictionary for a Typeable \(a\) constraint [Peyton Jones et al. 2016]—that is, we can use a runtime type test to discover the choice for \(a\). We would thus like to write the following function:

\[
\text{toDouble} :: T \rightarrow \text{Double}
\]

\[
\text{toDouble} \ (\text{MkTF} \ a) = \ldots
\]

but that does not quite work. The problem is that \(F\) is not injective. The \(a\) in that pattern type annotation need not be the same one packed into the existential type variable by \(\text{MkTF}\), and GHC rightly considers such an \(a\) to be ambiguous\(^5\).

The only workaround available today is to change \(\text{MkTF}\) to take a proxy argument:

\[
\text{data } \text{Proxy} \ a = \text{Proxy} \quad \text{-- in GHC’s Data.Proxy}
\]

\[
\text{data } T \ where
\]

\[
\text{MkTF} :: \forall a. \text{Typeable } a \Rightarrow \text{Proxy } a \rightarrow F \ a \rightarrow T
\]

\[
\text{toDouble} \ (\text{MkTF} \ (\_ :: \text{Proxy } a) \ x) = \ldots
\]

The \(\text{Proxy}\) type stores no runtime information (at runtime, it is isomorphic to \(()\), but the type \(\text{Proxy } a\) carries the all-important type variable \(a\). All datatypes are injective, so we can use this proxy argument to bind the type variable \(a\) in a way that we could not do previously.

As with many other examples, this is once again unsatisfying: it is a shame that we have to modify the data constructor declaration just to deal with type variable binding.

\(^4\)This ordering arises because any type variable in a pattern signature is bound within the pattern.

\(^5\)If \(F\) were an injective type family, we could label it as such to fix the problem [Stolarek et al. 2015]. But here we assume that it is not.
3.7 Conclusion

In this section we have seen that pattern signatures allow us to bring into scope the existentially-bound type variables of a data constructor, but that doing so can be clumsy, and occasionally impossible. We need something better.

4 Visible type application in patterns

Consider again Elab from Section 3.4:

```haskell
data Elab where
  MkElab :: Show a ⇒ [Maybe (Tree (a, Int))] → Elab
```

and suppose we want to build a value of type `Elab` containing an empty list. We cannot write just `MkElab []` because that is ambiguous: we must fix the type at which `MkElab` is called so that the compiler can pick the right `Show` dictionary. We can use a type signature, but it is clumsy, just as the pattern signature was clumsy in Section 3.4:

```haskell
MkElab [] :: [Maybe (Tree (a, Int))]
```

It is much nicer to use visible type application and write `MkElab @a []`. So it is natural to ask whether we could do the same in patterns, like this:

```haskell
getE :: Elab → Int
getE (MkElab @a xs) = ...a ...
```

Here, we bind `a` directly, as a type-argument pattern all by itself, rather than indirectly via a pattern signature.

We call this visible type application in patterns, a dual of visible type application in the same way that a pattern signatures are a dual of type signatures. This section describes visible type application in patterns informally, while the next formalizes it.

This feature was first requested more than two years ago.\(^6\) Furthermore, binding type variables like this is useful for more than just disambiguation, as we will shortly see.

4.1 Examples

Visible type application in patterns immediately fixes the other problems of pattern signatures identified above. For example, in the GADT example of Section 3.4 we can write

```haskell
matchGM :: a ⇒ GM a ⇒ Bool
matchGM x (MkMaybe @b) = isJust @b x
```

and for the type-family example of Section 3.6 we write

```haskell
toDouble (MkTF @a x) = ...
```

4.2 Universal and existential variables

Visible type applications in patterns can be used for all the type arguments of a data constructor, whether existential or universal. As an example of the latter we may write

```haskell
main = do (Just @Int x) ← readMaybe’ fmap’ getLine
        putStrLn "Input was " + show x
```

as an alternative to

```haskell
main = do (Just :: (x :: Int)) ← readMaybe’ fmap’ getLine
         putStrLn $ "Input was " + show x
```

Visible type application in patterns considers the type of data constructor, exactly as written by the user. For example

```haskell
data G a b where
  G1 :: ∀b. Char → G Int b
  G2 :: ∀p q b. p → q → b → G (p, q) b
  G3 :: ∀p q a. (a ~ (p, q)) ⇒ p → q → b → G a b
  f :: G a Bool → Int
  f (G1 @Bool y) = ord y
  f (G2 @p @q @x y z) = 0
  f (G3 @p @q @a @Bool @x y z) = 1
```

In this definition
- `G1` has one type argument.
- `G2` has three type arguments, but we have chosen to match only the first two.
- `G3` is morally identical to `G2`, because of the equality, but it is written with four type arguments, and visible type application in patterns follows that specification.

4.3 Type aliases

In Section 3.5 we have seen that GHC currently restricts type variables to refer to type variables, but that this does not have to be the case. Similar questions arise in our function `f` above. Could we write this for `G3`?

```haskell
f (G3 @p @q @(p, q) @b @x @y @z) = 1
```

Instead of `a` we have written `(p, q)`, which is equal to `a`. And instead of `Bool` we have written `b`, thereby binding `b` to `Bool`.\(^7\)

\(^6\)https://ghc.haskell.org/trac/ghc/ticket/11350

\(^7\)At this point it is instructive to note that GHC does not fully support the definition of `G` above, which is why we cannot write `f` as above.
Given the ubiquity of equalities, it no longer seems to make sense to restrict what a scoped type variable can stand for, so we propose simply to drop the restriction. Doing so simplifies the specification and the implementation of both pattern signatures and visible type application in patterns. A GHC proposal by one of the authors [Breitner 2018] is underway. Relaxing the requirement also allows the user to use type variables as “local type synonyms” that stand for possibly long types:

\[
\text{processMap} :: \text{Map} \text{ Int} \ (\text{Maybe} \ (\text{Complex Type})) \to ... \\
\text{processMap} \ (m :: \text{Map} \text{ key value}) = ...
\]

5 Formal Specification

We give the first formal specification of a number of extensions to Haskell related to pattern matching and the scoping of type variables: annotations in patterns, scoped type variables, and type application syntax in patterns. This section builds up these specification step by step, starting with a specification of the language without these features.

Our specification does not cover let-bindings and declaration type signatures. We focus instead on pattern signatures, which is where our new contribution lies. The scoping of forall-bound type variables from declaration type signatures would be straightforward to add.

5.1 The baseline

We begin with a reduced model of Haskell98 terms, which knows nothing yet about scoped type variables nor type equalities (i.e., no GADTs). We also removed type class constraints. The syntax is given in Fig. 2, and the typing rules are in Fig. 3. In the typing rules, we use a convention where an over-bar indicates a list, optionally with a superscript index to indicate the iterator. Iterators are additionally annotated with length bounds, where appropriate.

The grammar includes separate metavariables for internal type variables \(a\) and user type variables \(c\). The former are type variables as propagated by the compiler, while the latter are type variables the user has written. It is as if internal type variables \(a\) are spelled with characters unavailable in source Haskell. This distinction becomes important in Section 5.5. The language also includes only annotated let-bindings; no let-generalization here. (The “generalization” you might spot in Rule LET is simply quantifying over the variables the user has lexically written in the type signature.) This keeps our treatment simple and avoids the challenges of type inference. Allowing full let-generalization and un-annotated lets changes none of the conclusions presented here.

The judgment \(\Gamma \vdash e : \tau\) indicates that the term \(e\) has type \(\tau\) in the context \(\Gamma\), where \(\Gamma\) is a list of term variables and their (possibly polymorphic) types. Data constructors are globally fixed in an initial context \(\Gamma_0\); it is assumed that any context \(\Gamma\) contains the global \(\Gamma_0\) binding data constructors.

The type-checking of possibly nested patterns, as they occur in a case statement, is offloaded to the judgment \(\Gamma \vdash p : \sigma \Rightarrow \Gamma',\) which checks that \(p\) is a pattern for a value of type \(\sigma\) and possibly binds new term variables, which are added to \(\Gamma\) and returned in the extended environment \(\Gamma'\). The auxiliary judgment \(\Gamma \vdash_0 p_i : \sigma_i \Rightarrow \Gamma',\) straightforwardly threads the environment through a list of such pattern typings.

These rules should be unsurprising, but provide a baseline from which to build.
5.2 Support for GADTs

Now we extend this language with support for GADTs, with their existential type variables and equality constraints. See Fig. 4. The term syntax is unchanged, but polytypes now can mention constraints, which can either be empty (and elided from this text), an equality between two monotypes, or a conjunction of constraints. We leave the possibility open for additional constraints, as indicated by the ellipsis.

The environment $\Gamma$ is extended with two new forms. First, we track the scope of type variables by adding $a : \ast$ to $\Gamma^\forall$. Second, we add constraints $Q$ to $\Gamma$, to indicate which constraints (bound by a GADT pattern match) are in scope. Conversely, constraints are proved by the $\Gamma \vdash Q$ entailment relation.

Support for GADTs can be seen in the new form of data constructor types, listed in Fig. 4. Note that the arguments to $T$ in the return type are no longer confined to be $\bar{a}$, the quantified type variables; instead they can be arbitrary monotypes. In addition, a constructor can include a constraint $Q$.

When a data constructor is used in an expression, then the type equalities must be satisfied in the current environment, as expressed by the new premise of Rule $\text{VarConQ}$ in Fig. 4. We see also that the type equalities in the environment can be used for implicit coercion, as expressed in the Rule Eq.

When pattern-matching a data constructor, Rule $\text{PatCon}$ brings the type variables $\bar{a}$ into scope, by extending $\Gamma$. We require that these bound variables are fresh with respect to other variables in scope, a requirement we can satisfy by $\alpha$-renaming if necessary. We also add the type equalities that we have learned to the environment—that is, the equivalence between the $\bar{e}_i$ from the data constructor’s type and the $\bar{e}_j$ from the pattern type.

Finally, we update Rule $\text{CaseTv}$ to prevent skolem escape and Rule $\text{LetTv}$ to track the internal variables brought into scope. Note that these variables are internal only—the user cannot write them in a program.

At this point, our type system is comparable in expressiveness to the specification given by Vytiniotis et al. [2011]. A notable difference is that we explicitly handle nested patterns. This is important, as in the presence of GADTs, the precise formulation of how nested pattern are type-checked matters. For example, consider:

**data** $G$ $a$ **where**

$G1 ::= G$ $\text{Bool}$

$G2 ::= G$ $a$

$f :: (G\ a, a, G\ a) \to \text{Bool}$

$f\ (G1, \text{True}, \_) = \text{False}$

$f\ (\_, \text{True}, G1) = \text{False}$

Here the first equation for $f$ is fine, but the second is not, because the pattern $\text{True}$ cannot match against an argument of type $a$ until after the constructor $G1$ has been matched—and matching in Haskell is left-to-right.

5.3 Treating universals and existentials uniformly

A technical contribution of this paper is that Rule $\text{PatCon}$ is simpler and more uniform than the one usually given [e.g. by Vytiniotis et al. 2011], in that it does not distinguish the universal and existential type variables of the data constructor. Instead, all the type variables are freshly bound, with the equalities $\bar{e}_i \sim \bar{e}_j$ linking them to the context. In particular, these equalities take the place of the substitution written in the previous Rule Con98.

However, there is a worry: pattern-matching involving GADTs lacks principal types, and hence usually requires a type signature (see Section 2.1). If we treat vanilla, non-GADT Haskell98 data types in the same way as GADTs, do we lose type inference for ordinary Haskell98 definitions? Specifically, Vytiniotis et al. [2011, Section 5.6.1] describe...
The typing rule expresses this through the premise
\[ \Gamma \vdash \sigma \leq \sigma' \]
which says that the expected type of the pattern is more general than the annotated type \( \sigma' \). Note that this relationship is backwards from the usual expected/actual relationship in typing because patterns are in a negative position. The subtleties of of polytype subtyping are well explored in the literature\(^5\) and need not derail our exploration here. However, note that Rule \( \text{PatSig}\) checks pattern \( p \) with the annotated type \( \sigma' \), not the more general \( \sigma \)—after all, the user has asked us to use \( \sigma' \).

5.5 Scoped type variables

Next, we add support for open pattern signatures that bring type variables into scope. We need not change any syntax. Only the typing rule for annotated patterns changes: we replace \( \text{PatSig} \) with \( \text{PatSigTv} \) and add \( \text{LetForall} \) in Fig. 6.

The \( \text{LetForall} \) rule allows programmers to bring variables \( \bar{c} \) into scope when an explicit \( V \) is mentioned in the source. Note that this rule does not do any implicit lexical generalization: echoing GHC’s behavior, if the user writes a \( V, \forall \alpha \) all new variables to be used in the type signature must be bound explicitly.

In Rule \( \text{PatSigTv} \), the last two premises are identical to those of its predecessor \( \text{PatSig} \). The first premise extracts the type variables \( \bar{c} \) that are free in the user-written type signature \( \sigma' \), but not already in scope in \( \Gamma \). The “not already in scope” part reflects the discussion of Section 3.1.

But what if \( \Gamma \) contains a binding, introduced by rule \( \text{PatCon} \), for a type variable that just happens to have the same name as one of the \( \bar{c} \) in a user-written signature? After all, the names of the type variables in \( \text{PatCon} \) are arbitrary internal names; they just need to be fresh. Our solution is simple: we take advantage of the difference between internal type variables and external ones. The user cannot accidentally capture an internal variable.

The second (top-right) premise of Rule \( \text{PatSigTv} \) is the most unusual. It brings the variables \( \bar{c} \) into scope, but then also assumes that each variable \( c \) equals some internal variable \( b \). (Nothing stops multiple elements of \( \bar{b} \) from being the same internal variable.) Strikingly, the \( \bar{b} \) are mentioned nowhere else in the rule. The existence of the \( \bar{b} \) in this premise essentially says that the \( \bar{c} \) are merely a renaming of existing in-scope internal variables. In practice, the \( \bar{b} \) are chosen in order to make the subtyping relationship \( \Gamma' \vdash \sigma \leq \sigma' \)

\(^5\)GHC’s current implementation of subtyping is described by Eisenberg et al. [2016], who also cite other relevant publications on the subject.
Patterns \[
\rho := x \mid K \widehat{\rho} \mid \rho :: \sigma
\]

This version supports type applications only in constructor patterns; we study pattern synonyms in Appendix A. The syntax and new typing rule are shown in Fig. 7. Rule PatConTyApp looks scary, but it just integrates the concepts seen in Rule PatSigTv into Rule PatCon. We have kept all the iteration indices to help the reader match up which lists are expected to have the same size.

Let us look at each premise separately:

- Once again, the type variables \(\overline{a}^j_i\) are those that occur in the explicit type patterns but are not yet in scope. These are treated like type variables in a pattern signature: they are brought into scope here, each as a short-hand for some internal type variable \(b_i\).
- The environment \(\Gamma\) is extended to \(\Gamma'\) and contains now the (internal) type variables \(\overline{a}^j_i\), the user-written scoped type variables \(\pi^j_l\), the type equations equating each \(q_j\) to its internal type variable \(b_i\), the GADT equalities \(\overline{v}_i \sim \pi^j_i\), and the constraint \(Q\) captured by \(K\).
- The type patterns are checked against the types they match against. In contrast to pattern signatures, we use type equality here (\(~\) ), not the subtyping relation (\(\leq\)): no types involved can be polytypes, and so the subtyping relation degenerates to type equality.

As written here, the rule requires a type application for each type variable (note that the \(\widehat{\rho}\) use the same indices as the quantified type variables \(\overline{a}^j_i\) in \(K\’s\) type). However, we can weaken this requirement simply by dropping some \(\tau^j_s\) from both the conclusion and the relevant premises.

Just as in Rule SigPattTv:

- The \(b_i\) are mentioned nowhere else in the rule; instead, they are fixed such that the equality constraints for the \(\tau^j_s\) are entailed by \(\Gamma'\).
- The rule requires that each user-written type variable stands for an internal variable, but this choice could readily be changed by writing \(\overline{v}_i\) instead of \(\overline{b}_i\) in the definition of \(\Gamma'\).

5.7 Conclusion

Through the incremental building of rules, we can see precisely how the new feature of explicit binding sites for type variables fits into the existing typing framework. We have also explored two further extensions:

- Allowing type application in patterns headed by pattern synonym [Pickering et al. 2016]. Our framework extends well in this new context, offering no surprises (Appendix A).
- Incorporating explicit binding sites for type variables in the patterns of a \(\lambda\)-expression. This is slightly subtler (though the end result adds only one, simple typing rule), but is relegated to the appendix because it requires reasoning about bidirectional type checking.
Bringing all the necessary context into scope would take us too far afield here (Appendix B).

6 Alternative approaches

6.1 Universals vs. existentials

Type theorists are wont to separate quantified type variables in data constructors into two camps: universals and existentials. Here is a contrived but simple example:

\[
\text{data } \text{UnivEx } a \text{ where} \\
\quad \text{MkUE} :: \forall a \ b. a \rightarrow b \rightarrow \text{UnivEx } a \\
\quad \text{matchUE} :: \forall a. \text{UnivEx } a \rightarrow \ldots \\
\quad \text{matchUE (MkUE } x \ y) = \ldots
\]

In the constructor \text{MkUE}, the variable \( a \) is universal (it is fixed by the return type \( \text{UnivEx } a \)) while \( b \) is existential (it is not fixed by the result type). When we match on \text{MkUE} in \text{matchUE}, we might want to bind \( b \), as it is first brought into scope by the match. However, we never need to match \( a \), as it is already in scope from \text{matchUE}’s type signature.

An alternative design for type applications in patterns is to allow matching only existentials in pattern matches, thus:

\[
\quad \text{matchUE} :: \forall a. \text{UnivEx } a \rightarrow \ldots \\
\quad \text{matchUE (MkUE } @b x y) = \ldots
\]

Indeed, this forms the main payload of the original GHC proposal for binding type variables [Suarez 2017]. This design is attractive because the bindings would be concise: only those variables that need to be bound would be available. However, there are two distinct drawbacks:

- Given the complexity of Haskell, it can be hard to specify the difference between universals and existentials: Clearly, \( a \) is universal in the constructor \text{MkUE} above. But what if its type were \text{MkUE} :: \forall a \ b. a \rightarrow b \rightarrow \text{UnivEx } (\text{Id } a), \text{where } \text{Id } a \text{ is a type synonym? An injective type family? If we add } a \sim b \text{ to the constraints of } \text{MkUE}, \text{then } b \text{ is also fixed by the result type—does that make it a universal?}

The question of whether the value of a type variable is fixed by the return type depends on how smart the compiler is, and any specification would have to draw an arbitrary line. In the end, this would leave our users just very confused.

- When using a data constructor in an expression, the caller is free to instantiate both universals and existentials. Indeed, universals and existentials are utterly indistinguishable in expressions. That means that one might write \text{MkUE } @\text{Int } @\text{Bool } 5 \text{ True} \text{ in an expression. If we could match against only existentials in patterns, though, we would have to write a pattern } \text{MkUE } @b x y, \text{remembering to skip the universal } a. \text{This would both be confusing to users and weaken the ergonomics of pattern matching, whose chief virtue is that deconstructing a datatype resembles closely the syntax of constructing one.}

We thus prefer not to differentiate universals and existentials in this way.

6.2 The type-lambda approach

A plausible alternative approach to adding scoped type variables is to take a hint from System F, the explicitly-typed polymorphic lambda calculus [Girard 1990]. In System F, a type lambda, written “\( \Lambda \)”, binds a type variable, just as a term lambda, written “\( \lambda \)”, binds a term variable. For example:

\[
\begin{align*}
\text{id} : & \forall a. a \rightarrow a \\
\text{id} & = \lambda x. x
\end{align*}
\]

A term \( \Lambda a. e \) has type \( \forall a. t \), for some type \( t \), just as a term \( \lambda x. e \) has type \( \tau_1 \rightarrow \tau_2 \). Hence, a very natural idea is to bind a source-language type variable with a source-language type lambda. This “the type-lambda approach” is the one adopted by SML 97 [Milner et al. 1997]. In SML one can write:

\[
\begin{align*}
\text{fun } & \text{'a prefix } (x : \text{'a}) \text{ yss} = \\
& \text{let fun } xcons (ys : \text{'a list}) = x :: ys \text{ in} \\
& \text{map } xcons \text{ yss}
\end{align*}
\]

Here, “\('a\)” following the keyword \text{fun} is the binding site of an (optional) type parameter of \text{prefix}; it scopes over the patterns of the definition and its right hand side.

Just as Haskell has implicit quantification in type signatures, SML allows the programmer to introduce implicit type lambdas. This definition is elaborated into the previous one:

\[
\begin{align*}
\text{fun } & \text{prefix } (x : \text{'a}) \text{ yss} = \\
& \text{let fun } xcons (ys : \text{'a list}) = x :: ys \text{ in} \\
& \text{map } xcons \text{ yss}
\end{align*}
\]

The language definition gives somewhat intricate rules to explain how to place the implicit lambdas. For example:

\[
\text{fun } f \ x = \ldots (\text{fun } y. \text{'}a \Rightarrow y) \ldots
\]

Where is the type lambda that binds the type variable ‘\( a \)?’ In SML one cannot answer that question without knowing both what the “....” is, and the context for the definition \text{fun} \( f \). Roughly speaking, the type lambda for an implicitly-scoped type variable ‘\( a \)” is placed on the innermost function definition that encloses all the free occurrences of ‘\( a \)’. The rule [Milner et al. 1997] is only one informal, albeit carefully worded, paragraph; the formal typing rules assume that a pre-processing pass has inserted an explicit binding for every type variable that is implicitly bound by the above rule.

The type-lambda approach explicitly connects lexical scoping and quantification. In contrast, our approach decouples the two, by treating a lexically scoped type variable merely as an alias for a type (or type variable).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1319880, Grant No. 1521539, and Grant No. 1704041.
References


Pattern synonyms \( \in \mathcal{P} \),
Expressions \( e := \ldots \mid \mathcal{P} \),
Patterns \( p := \ldots \mid \mathcal{P} \) @ \( \bar{p} \)
Pattern synonym types \( \Gamma_1 : \mathcal{P} : \forall a. Q_r \Rightarrow \forall b. Q_p \Rightarrow \bar{a} \rightarrow \tau \)

\[
\begin{align*}
\Gamma &\vdash e : \tau \\
(P : \forall a. Q_r \Rightarrow \forall b. Q_p \Rightarrow \bar{a} \rightarrow \tau) &\in \Gamma \\
\theta &= \{ a \mapsto \tau \} \{ b \mapsto \tau' \} \\
\Gamma \vDash \theta(Q_r \land Q_p) \\
\Gamma \vdash P : \theta(v) \\
\end{align*}
\]

Expression typing

\[
\Gamma_{\tau \in \mathcal{P}} p : \sigma \Rightarrow \Gamma'\quad \text{Pattern typing}
\]

\[
\begin{align*}
(P : \forall a. Q_r \Rightarrow \forall b. Q_p \Rightarrow \bar{a} \rightarrow \tau) &\in \Gamma \\
\Gamma' &= \Gamma, \bar{a} : \tau, a_i \sim \nu'_j, \bar{a}_i \# \text{dom}(\Gamma) \\
\bar{a}_i &\in \text{ftv}(\tau_i, \tau_i') \setminus \text{dom}(\Gamma) \\
\bar{b}_j &\# \text{dom}(\Gamma) \\
\Gamma'' &= \Gamma', \bar{b}_j : \ast, \bar{c}_j : \ast, c_i \sim \bar{b}_j, \nu_j, Q_p \\
\Gamma'' &\vdash \nu_j \sim a_i \\
\Gamma''(\nu_j) \quad \bar{c}_j &\in \text{dom}(\Gamma) \\
\Gamma'' &\vdash \nu_j \sim \bar{b}_j \\
\Gamma'' &\vdash \nu_j \Gamma_p : \theta(\sigma_k)^k \Rightarrow \Gamma''' \\
\Gamma \vdash \bar{a}_i, \bar{b}_j \Rightarrow \Gamma'''' \\
\end{align*}
\]

PatSyn

\[
\begin{align*}
\Gamma_{\tau \in \mathcal{P}} P : \bar{a}_i \# \text{dom}(\Gamma) \\
\Gamma &\vdash a_i \Rightarrow \Gamma'''' \\
\end{align*}
\]

\[\text{Figure 8. Support for pattern synonyms}\]

A Pattern synonyms

Pattern synonym [Pickering et al. 2016] types differentiate universal type variables from existential type variables, and thus add superficial complexity to our rules. Happily, this complexity really is just superficial—the underlying mechanism we have laid out applies well in this new scenario. See Fig. 8.

A pattern synonym type has two contexts: the \( Q_r \) required constraint and the \( Q_p \) provided constraint. Using a pattern synonym in a pattern requires that the required constraint already be provable; in contrast, the provided constraint may be assumed in the pattern’s right-hand side. The provided constraint is the analogue of the constraint in a data constructor’s type, while the required constraint has no data constructor analogue. An example can help to demonstrate. If we define pattern \( \text{Three} = 3 \) and then write

\[
\begin{align*}
f &:: T \rightarrow \ldots \\
f \text{ Three} &= \ldots \\
\end{align*}
\]

we must be able to prove both that \( \text{Num} T \) and \( \text{Eq} T \) hold before the pattern match can type-check. Accordingly, the pattern \( \text{Three} \) gets type \( \forall a. (\text{Eq} a, \text{Num} a) \Rightarrow a \), where the constraint listed is the required one. (When only one constraint is written, we understand that it is the required one, not the provided one.)

Because of the two contexts, our old Rule \( \text{VarConQ} \) is insufficient to deal with pattern synonyms used in an expression. We thus introduce the straightforward Rule \( \text{Syn} \).

Corresponding to the two contexts, a pattern synonym has two sets of type variables, the universals \( \bar{a}_i \) and the existentials \( \bar{b}_j \). The universals are those that are mentioned in the result type \( \nu \); they can be fixed by comparing \( \nu \) with \( \tau \), the known type of the pattern10. In contrast, the existentials are simply brought into scope by the pattern; they match no other known type (except as entailed by the constraints \( Q_p \)). Let us work premise-by-premise through Rule PatSyn.

- We first look up the pattern synonym type in the environment.
- We then bring the universals \( \bar{a}_i \) into scope, allowing each universal variable \( a_i \) to equal some other type \( \nu'_j \). We can think of these \( \nu'_j \) as the instantiations for the variables \( a_i \).
- Under these assumptions (that is, after instantiating universals with arbitrary monotypes \( \nu'_j \)), we must ensure that the expected type of the pattern \( \tau \) matches the declared result type of the pattern synonym \( \nu \). We also must ensure that the required constraint \( Q_r \) indeed holds.
- Then, we collect the newly scoped type variables \( \bar{a}_i \).
- We extend the context yet again, bringing the existentials \( \bar{b}_j \) into scope, introducing the user-written scoped type variables \( \bar{c}_j \), asserting that each \( c_j \) is actually a renaming of an internal variable \( b_j \), and assuming the provided constraint \( Q_p \). (As before, we could reasonably change the \( b_j \) to a \( \nu'_j \) to allow scoped type variables to stand for types instead of variables.)
- We now check that the type patterns corresponding to universals \( \bar{a}_i \) indeed match the universals \( \bar{a}_i \).
- We also check that the type patterns corresponding to existentials \( \bar{b}_j \) match those existentials \( \bar{b}_j \).
- Finally, we use this extended context to check the pattern arguments \( \bar{p}_k \).

Though there are lots of details here, the only new piece is the check for the required constraint \( Q_r \). This new check fits nicely within our framework.

---

10There are some subtleties around the difference between universals and existentials. Pickering et al. [2016, Section 6.3] explore these challenges.
B  Binding type variables in function definitions

B.1 Motivation and examples

It is relatively easy to extend the ideas in our paper to allow type variable binding in function definitions. We can motivate this idea using an early example, repeated here:

```
prefix :: ∀a. a → [ [a] ] → [ [a] ]
prefix x yss = map xcons yss
  where xcons :: [a] → [a]
xcons ys = x : ys
```

The type signature includes a ∀ which brings the type variable a into scope in the body of prefix. We find this syntax design awkward however: a traditional reading of the syntax of ∀ tells us that a is in scope in a → [ [a] ] → [ [a] ] and nowhere else. Indeed, we should be able to α-vary the name a in that type signature (and only that type signature) without any external effect. Yet, it is not so, with a’s scope extending beyond that type signature.

Instead, we find it would be more natural (if, admittedly, a bit more verbose) to write

```
prefix @a x yss = ...
```

explicitly binding the type variable a in a pattern.

**GeneralizedNewtypeDeriving** This issue is not always so simple, however. Consider GHC’s implementation of the extension GeneralizedNewtypeDeriving, which uses coerce [Breitner et al. 2016] to translate the implementation of a class method from one type to another. Here is an example:

```
newtype MyList a = MkMyList [a]
deriving Functor

How should we write our derived Functor instance? A first attempt might look like this:

```
instance Functor MyList where
  fmap = coerce (fmap @[[]])
```

This, unfortunately, fails to compile. The problem is that the use of coerce is ambiguous. GHC can determine that the argument to coerce has type (a0 → b0) → [a0] → [b0] (the lists in there come from the use of @[[]]) and that the result must have type (a → b) → MyList a → MyList b.

However, GHC has no way to know that the undetermined a0 and b0 in coerce’s argument’s type should be the expected a and b in the result type. Perhaps GHC could guess this correspondence, but GHC refuses to make guesses.

The way to get this to work is to bring a and b into scope, like this:

```
instance Functor MyList where
  fmap :: ∀a. b. (a → b) → MyList a → MyList b
  fmap = coerce (fmap @[[]] @a @b)
```

Expressions  

```
e ::= ... | λc.e
```

![Figure 9. Binding type variables in λ-expressions](image)

By writing a type signature for fmap (using the InstanceSigs extension), we can bring a and b into scope to use as visible type arguments to the coerced fmap. As with the examples in our motivation section, this workaround is infelicitous, requiring enabling yet another extension and writing out an otherwise-redundant type, just to bind type variables.

With the ability to explicitly bind type variables, this instance is simplified to become

```
instance Functor MyList where
  fmap @a @b = coerce (fmap @[[]] @a @b)
```

**RankNTypes** Further motivation arises in the context of higher-rank types [Peyton Jones et al. 2007]. Suppose we have

```
hr :: (∀a. F a → F a) → ...
```

where F is a non-injective type family. At a call site of hr, ...
hr (λx → ...)..., there is no way to bind the type variable a, which is used only as the argument to non-injective type family. Currently, the only workaround here is to add a Proxy argument to hr’s argument type, allowing the client of hr to bind the type variable in a pattern annotation on the proxy argument.

Instead, our design allows users to bind the type variable explicitly, with ...
hr (λa x → ...)....

B.2 Specification

The ability to bind type variables makes sense only when we know the type of the function we are defining; this type tells us how the type variable relates to the other parameters and the result of the function. Note that the analogue of this requirement is always true for constructor patterns, where we can look up the type of the constructor in the environment. We thus must proceed by considering a bidirectional approach to type-checking, where explicitly track whether we are inferring a type or checking one. Specifically, we will build on one author’s previous work [Eisenberg et al. 2016], which sets the stage for our new extension here. Readers may want to consult Fig. 8 of that previous work in order to understand our new contribution.

Our addition to the previous 𝑖_{sl}^∗ judgment is in Fig. 9. The 𝑖_{sl}^∗ judgment checks an expression against a polytype containing outermost binders. The original judgment comprises only one rule, which skolemizes all these variables into fresh
type constants. With our addition, it is understood that the original rule, termed SB_DeepSkol, fires only when the new one does not apply. Our new rule simply brings $c$ into scope as an alias for the internal variable $a$ from the polytype we are checking against.

Incredibly, that’s it! No other changes are necessary. One of the claims in the previous work is that the design therein is flexible and extensible. Our success here is further evidence of that claim.

There are a few subtleties of this approach, of course:

**Type variables vs. types** The grammar extension and Rule SB_DTyAbs both require that the $\lambda$-expression binds a variable, never a full type. This contrasts with our approach for binding type variables in constructor patterns. The reason for the difference here is that, while we could mimic the earlier approach here, we would find that the type pattern would always just be a variable. In the case of $\lambda$-expressions, there is no analogue of the universal variables of data constructor types. Furthermore, any context $Q$ in the polytype being checked against will come into scope after the bound type variable, so there is no possibility of such a context entailing an equality between the bound variable and some other type. For similar reasons, there is no incentive to use an equality assumption in place of substitution in Rule SB_DTyAbs. We thus keep the rule simple by straightforwardly requiring the binding of only variables, and not full types.

**Comparison against declarative specification** Since we are working in the context of Eisenberg et al. [2016], we should consider what the implications are for the comparison of our extension of System SB against the declarative System B of that work. Happily, all we need to do to keep System B in sync with System SB is add an analogous rule to the “checking” judgment of System B: the fact that the judgment works only with specified polytypes (never generalized ones, in the terminology of the previous work) means that we need not worry about the effects of implicit generalization here.

**Deep skolemization** Unfortunately, the new system is not quite as expressive as we would like. Rule SB_DeepSkol performs deep skolemization [Peyton Jones et al. 2007, Section 4.6], meaning that any quantified type variables—even those to the right of arrows—are skolemized. Once these are skolemized, they are unavailable in later applications of SB_DTyAbs. Let us examine an example:

$$ \langle \lambda c1 \ x \ y \rightarrow \ldots \rangle :: \forall a1. a1 \rightarrow \forall a2. a2 \rightarrow \ldots $$

In this example, we are checking a $\lambda$-expression binding type variables against a polytype $\forall a1. a1 \rightarrow \forall a2. a2 \rightarrow \ldots$. First, SB_DTyAbs applies, binding $c1$ to be an alias for $a1$. Then, however, SB_DeepSkol applies, which will skolemizes $a2$, removing an opportunity to bind it. Accordingly, while the example above is accepted by our addition, this one would be rejected:

$$ \langle \lambda c1 \ x @c2 \ y \rightarrow \ldots \rangle :: \forall a1. a1 \rightarrow \forall a2. a2 \rightarrow \ldots $$

The difference here is that we try to bind $c2$ as an alias to $a2$. This fails because $a2$ has been skolemized before we can apply SB_DTyAbs.

One could easily wonder: why do deep skolemization here? This is answered in the last paragraph of Section 6.1 of the extended version of Eisenberg et al. [2016]—essentially, doing shallow skolemization would work but would make our ideas harder to implement. Currently, we have no non-contrived examples where this extra work is necessary, and so we leave the deep skolemization rule as it is.

---

11The previous work builds up its type systems by contrasting syntax-directed type systems against declarative specifications. The current paper focuses only on the syntax-directed approach.