**Overcoming Software**

# Basic Type Level Programming in Haskell

26 Apr 2017

Dependently typed programming is becoming all the rage these days. Advocates are talking about all the neat stuff you can do by putting more and more information into the type system. It's true! Type level programming gives you interesting new tools for designing software. You can guarantee safety properties, and in some cases, even gain performance optimizations through the use of these types.

I'm not going to try and sell you on these benefits – presumably you've read about something like the dependently typed neural networks, or about Idris's encoding of network protocols in the type system. If you're not convinced, then this isn't the right article for you. If you *are* interested, and have some familiarity with Haskell, Elm, F#, or another ML-family language, then this article will be right up your alley.

## The Basic Types

So let's talk about some basic types. I'm going to stick with the *real* basic types here: no primitives, just stuff we can define in one line in Haskell.

```
data Unit = MkUnit

data Bool = True | False
```

This code block defines two new types: `Unit` and `Bool`. The `Unit` type has one *constructor*, called `MkUnit`. Since there's only one constructor for this type, and it takes no parameters, there is only one value of this type. We call it `Unit` because there's only one value.

`Bool` is a type that has two *constructors*: `True` and `False`. These don't take any parameters either, so they're kind of like constants.

What does it mean to be a type? A type is a way of classifying things. Things – that's a vague word. What do I mean by 'things'?

Well, for these simple types above, we've already seen all their possible values – we can say that `True` and `False` are members of the type `Bool`. Furthermore, `1`, `'a'`, and `Unit` are *not* members of the type `Bool`.

These types are kind of boring. Let's look at another type:

```
data IntAndChar = MkIntAndChar Int Char
```

This introduces a type `IntAndChar` with a single *constructor* that takes two arguments: one of which is an `Int` and the other is a `Char`. Values of this type look like:

```
theFirstOne = MkIntAndChar 3 'a'
theSecond   = MkIntAndChar (-3) 'b'
```

`MkIntAndChar` looks a lot like a function. In fact, if we ask GHCi about it's type, we get this back:

```
λ> :t MkIntAndChar
MkIntAndChar :: Int -> Char -> IntAndChar
```

`MkIntAndChar` is a function accepting an `Int` and a `Char` and finally yielding a value of type `IntAndChar`.

So we can construct values, and values have types. Can we construct types? And if so, what do they have?

## The Higher Kinds

Let's hold onto our intuition about functions and values. A function with the type `foo :: Int -> IntAndChar` is saying:

> Give me a value with the type `Int`, and I will give you a value with the type `IntAndChar`.

Now, let's lift that intuition into the type level. A *value* constructor accepts a *value* and yields a *value*. So a *type* constructor accepts a *type* and yields a *type*. Haskell's type variables allow us to express this.

Let's consider everyone's favorite sum type:

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

Here, we declare a *type* `Maybe`, with two data constructors: `Just`, which accepts a value of type `a`, and `Nothing`, which does not accept any values at all. Let's ask GHCi about the type of `Just` and `Nothing`!

```
λ> :t Just
Just :: a -> Maybe a
λ> :t Nothing
Nothing :: Maybe a
```

So `Just` has that function type – and it looks like, whatever type of value we give it, it becomes a `Maybe` of that type. `Nothing`, however, can conjure up whatever type it wants, without needing a value at all. Let's play with that a bit:

```
λ> let nothingA = Nothing :: Maybe a
λ> let nothingInt = Nothing :: Maybe Int
λ> let nothingChar = Nothing :: Maybe Char
λ> nothingInt == nothingChar

\<interactive\>:27:15: error:
      • Couldn't match type ‘Char’ with ‘Int’
        Expected type: Maybe Int
          Actual type: Maybe Char
      • In the second argument of ‘(==)’, namely ‘no
thingChar’
        In the expression: nothingInt == nothingChar
        In an equation for ‘it’: it = nothingInt ==
nothingChar
λ> nothingA == nothingInt
True
λ> nothingA == nothingChar
True
```

Woah – we get a type error when trying to compare `nothingInt` with `nothingChar`. That makes sense – `(==)` only works on values that have the same type. But then, wait, why does `nothingA` not complain when compared with `nothingInt` and `nothingChar`?

The reason is that `nothingA :: Maybe a` *really* means:

> I am a value of `Maybe a`, for any and all types `a` that you might provide to me.

So I’m seeing that we’re passing types to `Maybe`, in much the same way that we pass values to `MkIntAndChar`. Let’s ask GHCi about the type of `Maybe`!

```
λ> :type Maybe

\<interactive\>:1:1: error:
    • Data constructor not in scope: Maybe
    • Perhaps you meant variable ‘maybe’ (imported
from Prelude)
```

Well, it turns out that types don’t have types (kind of, sort of). Types have *kinds*. We can ask GHCi about the *kind* of types with the `:kind` command:

```
λ> :kind Maybe
Maybe :: * -> *
```

What is `*` doing there? Well, `*` is the *kind* of types which have *values*. Check this out:

```
λ> :kind Maybe
Maybe :: * -> *
```

So `Maybe` has the *kind* `* -> *`, which means:

> Give me a type that has values, and I will give you a type that has values.

Maybe you've heard about *higher kinded polymorphism* before. Let's write a data type that demonstrates that this means:

```haskell
data HigherKinded f a
    = Bare a
    | Wrapped (f a)
```

Haskell's kind inference is awfully kind on the fingers – since we *use* `f` applied to `a`, Haskell just knows that `f` must have the kind `* -> *`. If we ask GHCi about the *kind* of `HigherKinded`, we get back:

```
λ> :kind HigherKinded
HigherKinded :: (* -> *) -> * -> *
```

So `HigherKinded` is a type that accepts a *type* of *kind* `* -> *`, and a *type* of *kind* `*`, and returns a type of kind `*`. In plain, verbose English, this reads as:

> Give me two types: the first of which is a function that does not have values itself, but when given a type that *does* have values, it can have values. The second being a type that has values. Finally, I will return to you a type that can have ordinary values.

# Dynamically Kinded Programming

`*` reminds me of regular expressions – “match anything.” Indeed, `*` matches *any* type that has values, or even types that are only inhabited by the infinite loop:

```
λ> data Void
λ> :kind Void
Void :: *
```

We don’t provide any ways to construct a void value, yet it still has kind `*`.

In the same way that you can productively program at the value level with dynamic *types*, you can productively program at the *type* level with *dynamic kinds*. And `*` is basically that!

Let’s encode our first type level numbers. We’ll start with the Peano natural numbers, where numbers are inductively defined as either `Zero` or the `Successor` of some natural number.

```
data Zero
data Succ a

type One = Succ Zero
type Two = Succ One
```

```haskell
type Three = Succ Two
type Four = Succ (Succ (Succ (Succ Zero)))
```

But this is pretty unsatisfying. After all, there's nothing that stops us from saying `Succ Bool`, which doesn't make any sense. I'm pretty sold on the benefits of types for clarifying thinking and preventing errors, so abandoning the safety of types when I program my types just seems silly. In order to get that safety back, we need to introduce more kinds than merely `*`. For this, we have to level up our GHC.

## Data Kinds

```haskell
{-# LANGUAGE DataKinds #-}
```

The `DataKinds` extension allows us to *promote* data constructors into type constructors, which also promotes their *type* constructors into *kind* constructors. To promote something up a level, we prefix the name with an apostrophe, or tick: `'`.

Now, let's define our *kind safe* type level numbers:

```haskell
data Nat = Zero | Succ Nat
```

In plain Haskell, this definition introduces a new type `Nat` with two *value* constructors, `Zero` and `Succ` (which takes a value of *type* `Nat` ). With the `DataKinds` extension, this *also* defines some extra new tidbits. We get a new *kind* `Nat` , which exists in a separate namespace. And we get two new types: a type constant `'Zero` , which has the *kind* `Nat` , and a type *constructor* `'Succ` , which accepts a *type* of *kind* `Nat` . Let's ask GHCi about our new buddies:

```
λ> :kind 'Zero
'Zero :: Nat
λ> :kind 'Succ
'Succ :: Nat -> Nat
```

You might think: that looks familiar! And it should. After all, the *types* look very much the same!

```
λ> :type Zero
Zero :: Nat
λ> :type Succ
Succ :: Nat -> Nat
```

Where it can be ambiguous, the `'` is used to disambiguate. Otherwise, Haskell can infer which you mean.

It's important to note that there are *no values* of type `'Zero` . The only *kind* that can have *types* that can have *values* is `*` .

We've gained the ability to construct some pretty basic types and kinds. In order to actually use them, though, we need a bit more power.

## GADTs

```
{-# LANGUAGE GADTs #-}
```

You may be wondering, "What does GADT stand for?" Richard Eisenberg will tell you that they're Generalized Algebraic Data Types, but that the terminology isn't helpful, so just think of them as Gadts.

GADTs are a tool we can use to provide extra type information by matching on constructors. They use a slightly different syntax than normal Haskell data types. Let's check out some simpler types that we'll write with this syntax:

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

The GADT syntax lists the constructors line-by-line, and instead of providing the *fields* of the constructor, we provide the *type signature* of the constructor. This is an interesting change – I just wrote out `a -> Maybe a`. That suggests, to me, that I can make these whatever type I want.

```
data IntBool a where
    Int :: Int -> IntBool Int
    Bool :: Bool -> IntBool Bool
```

This declaration creates a new type `IntBool`, which has the *kind* `* -> *`. It has two constructors: `Int`, which has the type `Int -> IntBool Int`, and `Bool`, which has the type `Bool -> IntBool Bool`.

Since the constructors carry information about the resulting type, we get bonus information about the type when we pattern match on the constructors! Check this signature out:

```
extractIntBool :: IntBool a -> a
extractIntBool (Int _)   = 0
extractIntBool (Bool b)  = b
```

Something *really* interesting is happening here! When we match on `Int`, we know that `IntBool a ~ IntBool Int`. That `~` tilde is a symbol for *type equality*, and introduces a *constraint* that GHC needs to solve to type check the code. For this branch, we know that `a ~ Int` , so we can return an `Int` value.

We now have enough power in our toolbox to implement everyone's favorite example of dependent types: length indexed vectors!

## Vectors

Length indexed vectors allow us to put the length of a list into the type system, which allows us to *statically* forbid out-of-bounds errors. We have a way to *promote* numbers into the type level using `DataKinds` , and we have a way to provide bonus type information using `GADTs` . Let's combine these two powers for this task.

I'll split this definition up into multiple blocks, so I can walk through it easily.

```
data Vector (n :: Nat) a where
```

We're defining a *type* `Vector` with *kind* `Nat -> * -> *` . The first type parameter is the length index. The second type parameter is the type of values contained in the vector. Note that, in order to compile something with a *kind* signature, we need…

```
{-# LANGUAGE KindSignatures #-}
```

Thinking about types often requires us to think in a logical manner. We often need to consider things inductively when constructing them, and recursively when destructing them. What is the base case for a vector? It's the empty vector, with a length of `Zero` .

```
VNil :: Vector 'Zero a
```

A value constructed by `VNil` can have any type `a` , but the length is always constrained to be `'Zero` .

The inductive case is adding another value to a vector. One more value means one more length.

```
VCons :: a -> Vector n a -> Vector ('Succ n) a
```

The `VCons` constructor takes two values: one of type `a` , and another of type `Vector n a` . We don't know how long the `Vector` provided is – it can be any `n` such that `n` is a `Nat` ural number. We *do* know that the *resulting* vector is the `Succ` essor of that `n` umber, though.

So here's the fully annotated and explicit definition:

```
data Vector (n :: Nat) (a :: *) where
    VNil :: Vector 'Zero a
    VCons :: a -> Vector n a -> Vector ('Succ n) a
```

Fortunately, Haskell can infer these things for us! Whether you use the above explicit definition or the below implicit definition is a matter of taste, aesthetics, style, and documentation.

```
data Vector n a where
    VNil :: Vector Zero a
    VCons :: a -> Vector n a -> Vector (Succ n) a
```

Let's now write a `Show` instance for these length indexed vectors. It's pretty painless:

```
instance Show a => Show (Vector n a) where
    show VNil          = "VNil"
    show (VCons a as)  = "VCons " ++ show a ++ " ("
++ show as ++ ")"
```

That `n` type parameter is totally arbitrary, so we don't have to worry about it too much.

## The Vector API

As a nice exercise, let's write `append :: Vector n a -> Vector m a -> Vector ??? a`. But wait, what is `???`

going to be? It needs to represent the *addition* of these two natural numbers. Addition is a *function*. And we don't have type functions, right? Well, we do, but we have to upgrade our GHC again.

```
{-# LANGUAGE TypeFamilies #-}
```

For some reason, functions that operate on types are called type families. There are two ways to write a type family: open, where anyone can add new cases, and closed, where all the cases are defined at once. We'll mostly be dealing with closed type families here.

So let's figure out how to add two `Nat` ural numbers, at the type level. For starters, let's figure out how to add at at the value level first.

```
add :: Nat -> Nat -> Nat
```

This is the standard Haskell function definitions we all know and love. We can pattern match on values, write `where` clauses with helpers, etc.

We're working with an inductive definition of numbers, so we'll need to use recursion get our answer. We need a base case, and then the inductive case. So lets start basic: if we add 0 to any number, then the answer is that number.

```haskell
add Zero n = n
```

The inductive case asks:

> If we add the successor of a number ( `Succ n` ) to another number ( `m` ), what is the answer?

Well, we know we want to get to `Zero` , so we want to somehow *shrink* our problem a bit. We'll have to shift that `Succ` from the left term to the right term. Then we can recurse on the addition.

```haskell
add (Succ n) m = add n (Succ m)
```

If you imagine a natural number as a stack of plates, we can visualize the *addition* of two natural numbers as taking one plate off the top of the first stack, and putting it on top of the second. Eventually, we'll use all of the plates – this leaves us with `Zero` plates, and our final single stack of plates is the answer.

```haskell
add :: Nat -> Nat -> Nat
add Zero     n = n
add (Succ n) m = add n (Succ m)
```

Alright, let's promote this to the type level.

## Type Families

The first line of a type family definition is the signature:

```
type family Add n m where
```

This introduces a new *type function* `Add` which accepts two parameters. We can now define the individual cases. We can pattern match on type constructors, just like we can pattern match on value constructors. So we'll write the `Zero` case:

```
    Add 'Zero n = n
```

Next, we recurse on the inductive case:

```
    Add ('Succ n) m = Add n ('Succ m)
```

Ahh, except now GHC is going to give us an error.

```
      • The type family application ‘Add n ('Succ
m)’
          is no smaller than the instance head
        (Use UndecidableInstances to permit this)
      • In the equations for closed type family ‘Ad
d’
        In the type family declaration for ‘Add’
```

GHC is extremely scared of undecidability, and won't do *anything* that it can't easily figure out on it's own. `UndecidableInstances` is an extension which allows you to say:

> Look, GHC, it's okay. I know you can't figure this out. I promise this makes sense and will eventually terminate.

So now we get to add:

```
{-# LANGUAGE UndecidableInstances #-}
```

to our file. The type family definition compiles fine now. How can we test it out?

Where we used `:kind` to inspect the kind of types, we can use `:kind!` to *evaluate* these types as far as GHC can. This snippet illustrates the difference:

```
λ> :kind Add (Succ (Succ Zero)) (Succ Zero)
Add (Succ (Succ Zero)) (Succ Zero) :: Nat
λ> :kind! Add (Succ (Succ Zero)) (Succ Zero)
Add (Succ (Succ Zero)) (Succ Zero) :: Nat
= 'Succ ('Succ ('Succ 'Zero))
```

The first line just tells us that the result of `Add`ing two `Nat`ural numbers is itself a `Nat`ural number. The second line shows the actual result of evaluating the type level function. Cool! So now we can finally finish writing `append`.

```
append :: Vector n a -> Vector m a -> Vector (Add
n m) a
```

Let’s start with some bad attempts, to see what the types buy us:

```
append VNil rest = VNil
```

This fails with a type error – cool!

```
      • Could not deduce: m ~ 'Zero
        from the context: n ~ 'Zero
          bound by a pattern with constructor:
                     VNil :: forall a. Vector 'Zero
a,
                   in an equation for ‘append’
          at /home/matt/Projects/dep-types.hs:31:8-1
1
        ‘m’ is a rigid type variable bound by
          the type signature for:
            append :: forall (n :: Nat) a (m :: Na
t).
                      Vector n a -> Vector m a -> Ve
ctor (Add n m) a
          at /home/matt/Projects/dep-types.hs:30:11
        Expected type: Vector (Add n m) a
          Actual type: Vector 'Zero a
      • In the expression: VNil
        In an equation for ‘append’: append VNil res
t = VNil
      • Relevant bindings include
          rest :: Vector m a
            (bound at /home/matt/Projects/dep-types.
hs:31:13)
          append :: Vector n a -> Vector m a -> Vect
or (Add n m) a
            (bound at /home/matt/Projects/dep-types.
hs:31:1)
```

The error is kinda big and scary at first. Let's dig into it a bit.

GHC is telling us that it can't infer that `m` (which is the length of the second parameter vector) is equal to `Zero`. It knows that `n` (the length of the first parameter) is `Zero` because we've pattern matched on `VNil`. So, what values *can* we return? Let's replace the definition we have thus far with `undefined`, reload in GHCi, and inspect some types:

```
λ> :t append VNil
append VNil :: Vector m a -> Vector m a
```

We need to construct a value `Vector m a`, and we have been given a value `Vector m a`. BUT – we don't know what `m` is! So we have no way to spoof this or fake it. We have to return our input. So our first case is simply:

```
append VNil xs = xs
```

Like with addition of natural numbers, we'll need to have the inductive case. Since we have the base case on our first parameter, we'll want to try shrinking our first parameter in the recursive call.

So let's try another bad implementation:

```
append (VCons a rest) xs = append rest (VCons a x
s)
```

This doesn't really do what we want, which we can verify in the REPL:

```
λ> append (VCons 1 (VCons 3 VNil)) (VCons 2 VNil)
VCons 3 (VCons 1 (VCons 2 (VNil)))
```

The answer *should* be `VCons 1 (VCons 3 (VCons 2 VNil))` . However, our Vector type only encodes the *length* of the vector in the type. The sequence is not considered. Anything that isn't lifted into the type system doesn't get any correctness guarantees.

So let's fix the implementation:

```
append (VCons a rest) xs = VCons a (append rest x
    s)
```

And let’s reload in GHCi to test it out!

```
λ> :reload
[1 of 1] Compiling DepTypes         ( /home/matt/P
rojects/dep-types.hs, interpreted )

/home/matt/Projects/dep-types.hs:32:28: error:
    • Could not deduce: Add n1 ('Succ m) ~ 'Succ
(Add n1 m)
      from the context: n ~ 'Succ n1
        bound by a pattern with constructor:
                   VCons :: forall a (n :: Nat).
                            a -> Vector n a -> Vec
tor ('Succ n) a,
                 in an equation for ‘append’
        at /home/matt/Projects/dep-types.hs:32:9-2
0
      Expected type: Vector (Add n m) a
        Actual type: Vector ('Succ (Add n1 m)) a
    • In the expression: VCons a (append rest xs)
      In an equation for ‘append’:
          append (VCons a rest) xs = VCons a (appe
nd rest xs)
    • Relevant bindings include
        xs :: Vector m a (bound at /home/matt/Proj
ects/dep-types.hs:32:23)
        rest :: Vector n1 a
          (bound at /home/matt/Projects/dep-types.
hs:32:17)
        append :: Vector n a -> Vector m a -> Vect
or (Add n m) a
          (bound at /home/matt/Projects/dep-types.
hs:31:1)
Failed, modules loaded: none.
```

Oh no! A type error! GHC can’t figure out that `Add n (Succ m)` is the same as `Succ (Add n m)`. We can kinda see what went wrong if we lay the `Vector`, `Add` and `append` definitions next to each other:

```
data Vector n a where
    VNil :: Vector Zero a
    VCons :: a -> Vector n a -> Vector (Succ n) a

type family Add x y where
    Add 'Zero n = n
    Add ('Succ n) m = Add n ('Succ m)

append :: Vector n a -> Vector m a -> Vector (Add
n m) a
append VNil xs              = xs
append (VCons a rest) xs = VCons a (append rest x
s)
```

In `Vector`'s inductive case, we are building up a bunch of `Succ`s. In `Add`'s recursive case, we're tearing down the left hand side, such that the exterior is another `Add`. And in `append`s recursive case, we're building up the right hand side. Let's trace how this error happens, and supply some type annotations as well:

```
append (VCons a rest) xs =
```

Here, we know that `VCons a rest` has the type `Vector (Succ n) a`, and `xs` has the type `Vector m a`. We need to produce a result of type `Vector (Add (Succ n)` `m) a` in order for the type to line up right. We use `VCons a (append rest xs)`. `VCons` has a length value that is the `Succ` essor of the result of `append rest xs`, which should have the value `Add n m`, so the length there is `Succ (Add n m)`. Unfortunately, our result type needs to be `Add (Succ n) m`.

We know these values are equivalent. Unfortuantely, GHC cannot prove this, so it throws up it's hands. Two definitions, which are provably equivalent, are *structurally* different, and this causes the types and proofs to fail. This is a HUGE gotcha in type level programming – the implementation details matter, a lot, and they leak, *hard*. We can fix this by using a slightly different definition of `Add`:

```
type family Add x y where
    Add 'Zero n = n
    Add ('Succ n) m = 'Succ (Add n m)
```

This definition has a similar structure of recursion – we pull the `Succ` s out, which allows us to match the way that `VCons` adds `Succ` on top.

This new definition compiles and works fine.

## This Sucks

Agreed, which is why I'll defer the interested reader to this much better tutorial on length indexed vectors in Haskell. Instead, let's look at some other more interesting and practical examples of type level programming.

## Heterogeneous Lists

Heterogeneous lists are kind of like tuples, but they're defined inductively. We keep a type level list of the contents of the heterogeneous list, which let us operate safely on them.

To use ordinary Haskell lists at the type level, we need another extension:

```
{-# LANGUAGE TypeOperators #-}
```

which allows us to use operators at the type level.

Here's the data type definition:

```
data HList xs where
    HNil :: HList '[]
    (:::) :: a -> HList as -> HList (a ': as)

infixr 6 :::
```

The `HNil` constructor has an empty list of values, which makes sense, because it doesn't have any values! The `:::` construction operator takes a value of type `a`, an `HList` that already has a list of types `as` that it contains, and returns an `HList` where the first element in the type level list is `a` followed by `as`.

Let's see what a value for this looks like:

```
λ> :t 'a' ::: 1 ::: "hello" ::: HNil
'a' ::: 1 ::: "hello" ::: HNil
    :: HList '[Char, Int, String]
```

So now we know that we have a `Char`, `Int`, and `String` contained in this `HList`, and their respective indexes. What if we want to `Show` that?

```
λ> 'a' ::: 1 ::: "hello" ::: HNil

\<interactive\>:13:1:
    No instance for (Show (HList '[Char, Int, Stri
ng]))
      arising from a use of ‘print’
    In the first argument of ‘print’, namely ‘it’
    In a stmt of an interactive GHCi command: prin
t it
```

Hmm. We'll need to write a `Show` instance for `HList`. How should we approach this? Let's try something dumb first. We'll ignore all the contents!

```
instance Show (HList xs) where
    show HNil          = "HNil"
    show (x ::: rest)  = "_ ::: " ++ show rest
```

Ahah! this compiles, and it even works!

```
λ> 'a' ::: 1 ::: "hello" ::: HNil
_ ::: _ ::: _ ::: HNil
```

Unfortunately, it’s not very useful. Can we do better? We can!

## Inductive Type Class Instances

First, we’ll define the base case – showing an empty HList!

```
instance Show (HList '[]) where
    show HNil = "HNil"
```

This causes a compile error, requiring that we enable yet another language extension:

```
{-# LANGUAGE FlexibleInstances #-}
```

If you’re doing this in another file than the type family above, you'll also get an error about `FlexibleContexts`. It turns out that enabling `UndecidableInstances` implies `FlexibleContexts` for some reason. So let's throw that one on too, for good measure:

```
{-# LANGUAGE FlexibleContexts #-}
```

This compiles, and we can finally `show HNil` and it works out. Now, we must recurse!

The principle of induction states that:

1. We must be able to do something for the base case.
2. If we can do something for a random case, then we can do it for a case that is one step larger.
3. By 1 and 2, you can do it for all cases.

We've covered that base case. We'll assume that we can handle the smaller cases, and demonstrate how to handle a slightly large case:

```
instance (Show (HList as), Show a)
    => Show (HList (a ': as)) where
    show (a ::: rest) =
        show a ++ " ::: " ++ show rest
```

This instance basically says:

> Given that I know how to `Show` an `HList` of `as`, and I know how to `Show` an `a`: I can `Show` an `HList` with an `a` and a bunch of `as`.

```
λ> 'a' ::: 1 ::: "hello" ::: HNil
'a' ::: 1 ::: "hello" ::: HNil
```

## Further Exercises

Write an `aeson` instance for `HList`. It'll be similar to the `Show` instance, but require a bit more stuff.

## Extensible Records

There are a few variants on extensible records in Haskell. Here's a tiny implementation that requires yet more extensions:

```
{-# LANGUAGE PolyKinds        #-}
{-# LANGUAGE TypeApplications #-}

import GHC.TypeLits (KnownSymbol, symbolVal)
import Data.Proxy
```

This generalizes definitions for type variables, which allows for non-value type variables to have kind polymorphism. Type applications allow us to explicitly pass types as arguments using an `@` symbol.

First, we must define the type of our fields, and then our record:

```
newtype s >> a = Named a

data HRec xs where
    HEmpty :: HRec '[]
    HCons :: (s >> a) -> HRec xs -> HRec (s >> a
': xs)
```

The `s` parameter is going to be a type with the *kind* `Symbol`. `Symbol` is defined in `GHC.TypeLits`, so we need that import to do the fun stuff.

We'll construct a value using the `TypeApplications` syntax, so a record will look like:

```
λ> HCons (Named @"foo" 'a') (HCons (Named @"bar"
(3 :: Int)) HEmpty)

\<interactive\>:10:1: error:
    • No instance for (Show (HRec '["foo" >> Char,
"bar" >> Int]))
        arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: pr
int it
```

So, this type checks fine! Cool. But it does not Show, so we need to define a `Show` instance.

Those string record fields only exist at the type level – but we can use the `KnownSymbol` class to bring them back down to the value level using `symbolVal` .

Here’s our base case:

```
instance Show (HRec '[]) where
    show _ = "HEmpty"
```

And, when we recurse, we need a tiny bit more information. Let’s start with the instance head first, so we know what variables we need:

```
instance Show (HRec (s >> a ': xs)) where
```

OK, so we have a `s` type, which has the kind `Symbol` , an `a :: *` , and `xs` . So now we pattern match on that bad boy:

```
instance Show (HRec (s >> a ': xs)) where
    show (HCons (Named a) rest) =
```

OK, so we need to `show` the `a` value. Easy. Which means we need a `Show a` constraint tacked onto our instance:

```haskell
instance (Show a)
    => Show (HRec (s >> a ': xs)) where
    show (HCons (Named a) rest) =
        let val = show a
```

Next up, we need the key as a string. Which means we need to use `symbolVal`, which takes a `proxy s` and returns the `String` associated with the `s` provided that `s` is a `KnownSymbol`.

```haskell
instance (Show a, KnownSymbol s)
    => Show (HRec (s >> a ': xs)) where
    show (HCons (Named a) rest) =
        let val = show a
            key = symbolVal (Proxy :: Proxy s)
```

At this point, you're probably going to get an error like `No instance for 'KnownSymbol s0'`. This is because Haskell's type variables have a very limited scope by default. When you write:

```haskell
topLevelFunction :: a -> (a -> b) -> b
topLevelFunction a = go
  where
    go :: (a -> b) -> b
    go f = f a
```

Haskell interprets each type signature as it's own *scope* for the type variables. This means that the `a` and `b` variables in the `go` helper function are different type variables, and a more precise way to write it would be:

```
topLevelFunction :: a0 -> (a0 -> b0) -> b0
topLevelFunction a = go
  where
    go :: (a1 -> b1) -> b1
    go f = f a
```

If we want for type variables to have a scope similar to other variables, we need another extension:

```
{-# LANGUAGE ScopedTypeVariables #-}
```

Finally, we need to show the rest of the stuff!

```
instance (Show a, KnownSymbol s, Show (HRec xs))
    => Show (HRec (s >> a ': xs)) where
    show (HCons (Named a) rest) =
        let val = show a
            key = symbolVal (Proxy :: Proxy s)
            more = show rest
         in "(" ++ key ++ ": " ++ val ++ ") " ++ m
ore
```

This gives us a rather satisfying `Show` instance, now:

```
λ> HCons (Named @"foo" 'a') (HCons (Named @"bar"
(3 :: Int)) HEmpty)
(foo: 'a') (bar: 3) HEmpty
```

## Exercise:

Write an Aeson `ToJSON` instance for this `HRec` type which converts into a JSON object.

Bonus points: Write an Aeson `FromJSON` instance for the `HRec` type.

## Like what you read?

If you enjoyed this post, you should check out the book Thinking with Types by Sandy Maguire. It is an extensive manual on practical type-level programming in Haskell.

---

My blog is hosted on GitHub. If you'd like to leave a comment, report a problem, or contact me, then that's a fine place to do so.

- The rest of my posts
- My tutorials
- Haskell Consulting Services
- Check out my book - "Production Haskell"!

- Want to thank me for my posts?