

Notes for 'Thinking with Types: Type-level Programming in Haskell', Chapters 1–5

✍ March 18, 2020

🕒 A fifteen minute read

🏷 Tags: haskell, notes, programming

Haskell — with its powerful type system — has a great support for type-level programming and it has gotten much better in the recent times with the new releases of the GHC compiler. But type-level programming remains a daunting topic even with seasoned haskellers. *Thinking with Types: Type-level Programming in Haskell* by Sandy Maguire is a book which attempts to fix that. I've taken some notes to summarize my understanding of the same.

Contents

1. Introduction
2. Chapter 1. The Algebra Behind Types
 - 2.1. Isomorphisms and Cardinalities
 - 2.2. Sum, Product and Exponential Types
 - 2.3. The Curry-Howard Isomorphism
 - 2.4. Canonical Representations
3. Chapter 2. Terms, Types and Kinds
 - 3.1. The Kind System
 - 3.2. Data Kinds
 - 3.3. Promotion of Built-In Types
 - 3.4. Type-level Functions
4. Chapter 3. Variance
5. Chapter 4. Working with Types
6. Chapter 5. Constraints and GADTs
 - 6.1. Constraints
 - 6.2. GADTs
 - 6.3. Heterogeneous Lists
 - 6.4. Creating New Constraints
7. Conclusion

Introduction

- ▶ Type-level Programming (TLP) is writing programs that run at compile-time, unlike term-level programming which is writing programs that run at run-time.
- ▶ TLP should be used in moderation.

- TLP should be mostly used
 - for programs that are catastrophic to get wrong (finance, healthcare, etc).
 - when it simplifies the program API massively.
 - when power-to-weight ratio of adding TLP is high.
- Types are not a silver bullet for fixing all errors:
 - Correct programs can be not well-typed.
 - It can be hard to assign type for useful programs. e.g. `printf` from C.
- Types can turn possible runtime errors into compile-time errors.

Chapter 1. The Algebra Behind Types

Isomorphisms and Cardinalities

- *Cardinality* of a type is the number of values it can have ignoring bottoms. The values of a type are also called the *inhabitants* of the type.

```

1 data Void
2 -- no possible values. cardinality: 0
3 data Unit = Unit
4 -- only one possible value. cardinality: 1
5 data Bool = True | False
6 -- only two possible values. cardinality: 2

```

- Cardinality is written using notation: $|Void| = 0$
- Two types are said to be *Isomorphic* if they have same cardinality.
- An *isomorphism* between types *a* and *b* is a pair of functions *to* and *from* such that:

```

1 to :: a -> b
2 from :: b -> a
3 to . from = id
4 from . to = id

```

Sum, Product and Exponential Types

- Either *a b* is a *Sum* type. Its number of inhabitants is sum of the number of inhabitants of type *a* and *b* like so: $|a|$ possible values with the `Left` constructor and $|b|$ possible values with the `Right` constructor. Formally:

```

1 |Either a b| = |a| + |b|

```

- *(a, b)* is a *Product* type. Its number of inhabitant is the product of the number of inhabitants of types *a* and *b*. Formally:

```

1 |(a, b)| = |a| * |b|

```

- Some more examples:

```

1 |Maybe a| = |Nothing| + |Just a| = 1 + |a|
2 |[a]| = 1 + |a| + |a|^2 + |a|^3 + ...
3 |Either a Void| = |a| + 0 = |a|
4 |Either Void a| = 0 + |a| = |a|
5 |(a, Unit)| = |a| * 1 = |a|
6 |(Unit, a)| = 1 * |a| = |a|

```

► Function types are exponentiation types.

```

1 |a -> b| = |b|^|a|

```

For every value in domain a there can be $|b|$ possible values in the range b . And there are $|a|$ possible values in domain a . So:

```

1 |a -> b|
2   = |b| * |b| * ... * |b| -- (|a| times)
3   = |b|^|a|

```

► Data can be represented in many possible isomorphic types. Some of them are more useful than others. Example:

```

1  data TicTacToe1 a = TicTacToe1
2    { topLeft      :: a
3    , topCenter    :: a
4    , topRight     :: a
5    , middleLeft   :: a
6    , middleCenter :: a
7    , middleRight  :: a
8    , bottomLeft   :: a
9    , bottomCenter :: a
10   , bottomRight  :: a
11   }
12
13 |TicTacToe1 a|
14 = |a| * |a| * ... * |a| -- 9 times
15 = |a|^9
16
17 emptyBoard1 :: TicTacToe1 (Maybe Bool)
18 emptyBoard1 =
19   TicTacToe1 Nothing Nothing Nothing
20               Nothing Nothing Nothing
21               Nothing Nothing Nothing
22
23 -- Alternatively
24
25 data Three = One | Two | Three
26 data TicTacToe2 a =
27   TicTacToe2 (Three -> Three -> a)
28
29 |TicTacToe2 a| = |a|^(|Three| * |Three|)
30               = |a|^(3*3)
31               = |a|^9
32
33 emptyBoard2 :: TicTacToe2 (Maybe Bool)
34 emptyBoard2 =
35   TicTacToe2 $ const $ const Nothing

```

The Curry-Howard Isomorphism

- ▶ Every logic statement can be expressed as an equivalent computer program.
- ▶ Helps us analyze mathematical theorems through programming.

Canonical Representations

- ▶ Since multiple equivalent representations of a type are possible, the representation in form of sum of products is considered the canonical representation of the type. Example:

```

1 Either a (Either b (c, d)) -- canonical
2
3 (a, Bool) -- not canonical
4 Either a a
5 -- same cardinality as above but canonical

```

Chapter 2. Terms, Types and Kinds

The Kind System

- *Terms* are things manipulated at runtime. *Types* of terms are used by compiler to prove “things” about the terms.
- Similarly, *Types* are things manipulated at compile-time. *Kinds* of types are used by the compiler to prove “things” about the types.
- Kinds are “the types of the Types”.
- Kind of things that can exist at runtime (terms) is *. That is, kind of Int, String etc is *.

```

1 > :type True
2 True :: Bool
3 > :kind Bool
4 Bool :: *

```

- There are kinds other than *. For example:

```

1 > :kind Show Int
2 Show Int :: Constraint

```

- Higher-kinded types have (->) in their kind signature:

```

1 > :kind Maybe
2 Maybe :: * -> *
3 > :kind Maybe Int
4 Maybe Int :: *
5
6 > :type Control.Monad.Trans.Maybe.MaybeT
7 Control.Monad.Trans.Maybe.MaybeT
8   :: m (Maybe a) -> Control.Monad.Trans.Maybe.MaybeT m a
9 > :kind Control.Monad.Trans.Maybe.MaybeT
10 Control.Monad.Trans.Maybe.MaybeT :: (* -> *) -> * -> *
11 > :kind Control.Monad.Trans.Maybe.MaybeT IO Int
12 Control.Monad.Trans.Maybe.MaybeT IO Int :: *

```

Data Kinds

- -XDataKinds extension lets us create new kinds.
- It lifts data constructors into type constructors and types into kinds.

```

1 > :set -XDataKinds
2 > data Allow = Yes | No
3 > :type Yes
4 Yes :: Allow
5 -- Yes is data constructor
6 > :kind Allow -- Allow is a type
7 Allow :: *
8 > :kind 'Yes
9 'Yes :: Allow
10 -- 'Yes is a type too. Its kind is 'Allow.

```

- ▶ Lifted constructors and types are written with a preceding ' (called *tick*).

Promotion of Built-In Types

- ▶ -XDataKinds extension promotes built-in types too.
- ▶ Strings are promoted to the kind `Symbol`.
- ▶ Natural numbers are promoted to the kind `Nat`.

```

1 > :kind "hi"
2 "hi" :: GHC.Types.Symbol
3 -- "hi" is a type-level string
4 > :kind 123
5 123 :: GHC.Types.Nat
6 -- 123 is a type-level natural number

```

- ▶ We can do type level operations on Symbols and Nats.

```

1 > :m +GHC.TypeLits
2 GHC.TypeLits> :kind AppendSymbol
3 AppendSymbol :: Symbol -> Symbol -> Symbol
4 GHC.TypeLits> :kind! AppendSymbol "hello " "there"
5 AppendSymbol "hello " "there" :: Symbol
6 = "hello there"
7 GHC.TypeLits> :set -XTypeOperators
8 GHC.TypeLits> :kind! (1 + 2) ^ 7
9 (1 + 2) ^ 7 :: Nat
10 = 2187

```

- ▶ -XTypeOperators extension is needed for applying type-level functions with symbolic identifiers.
- ▶ There are type-level lists and tuples:

```

1  GHC.TypeLits> :kind '[ 'True ]
2  '[ 'True ] :: [Bool]
3  GHC.TypeLits> :kind '[1,2,3]
4  '[1,2,3] :: [Nat]
5  GHC.TypeLits> :kind '["abc"]
6  '["abc"] :: [Symbol]
7  GHC.TypeLits> :kind 'False ': 'True ': '[]
8  'False ': 'True ': '[] :: [Bool]
9  GHC.TypeLits> :kind '(6, "x", 'False)
10 '(6, "x", 'False) :: (Nat, Symbol, Bool)

```

Type-level Functions

- With the `-XTypeFamilies` extension, it's possible to write new type-level functions as closed type families:

```

1  > :set -XDataKinds
2  > :set -XTypeFamilies
3  > :{
4  | type family And (x :: Bool) (y :: Bool) :: Bool where
5  |   And 'True 'True = 'True
6  |   And _      _    = 'False
7  | :}
8  > :kind And
9  And :: Bool -> Bool -> Bool
10 > :kind! And 'True 'False
11 And 'True 'False :: Bool
12 = 'False
13 > :kind! And 'True 'True
14 And 'True 'True :: Bool
15 = 'True
16 > :kind! And 'False 'True
17 And 'False 'True :: Bool
18 = 'False

```

Chapter 3. Variance

- ▶ There are three types of *Variance* (τ here a type of kind $* \rightarrow *$):
 - ▶ Covariant: any function of type $a \rightarrow b$ can be lifted into a function of type $\tau a \rightarrow \tau b$.
Covariant types are instances of the `Functor` typeclass:

```
1 class Functor f where
2   fmap :: (a -> b) -> f a -> f b
```

- ▶ Contravariant: any function of type $a \rightarrow b$ can be lifted into a function of type $\tau b \rightarrow \tau a$.
Contravariant functions are instances of the `Contravariant` typeclass:

```
1 class Contravariant f where
2   contramap :: (a -> b) -> f b -> f a
```

- ▶ Invariant: no function of type $a \rightarrow b$ can be lifted into a function of type τa . Invariant functions are instances of the `Invariant` typeclass:

```
1 class Invariant f where
2   invmap :: (a -> b) -> (b -> a) -> f a -> f b
```

- ▶ Variance of a type τ is specified with respect to a particular type parameter. A type τ with two parameters a and b could be covariant wrt. a and contravariant wrt. b .
- ▶ Variance of a type τ wrt. a particular type parameter is determined by whether the parameter appears in positive or negative *positions*.
 - ▶ If a type parameter appears on the left-hand side of a function, it is said to be in a negative position. Else it is said to be in a positive position.
 - ▶ If a type parameter appears only in positive positions then the type is covariant wrt. that parameter.
 - ▶ If a type parameter appears only in negative positions then the type is contravariant wrt. that parameter.
 - ▶ If a type parameter appears in both positive and negative positions then the type is invariant wrt. that parameter.
 - ▶ positions follow the laws of multiplication for their *signs*.

a	b	a * b
+	+	+
+	-	-
-	+	-
-	-	+

- ▶ Examples:


```

1  newtype T1 a = T1 (Int -> a)
2  -- a is in +ve position, T1 is covariant wrt. a.
3  newtype T2 a = T2 (a -> Int)
4  -- a is in -ve position, T2 is contravariant wrt. a.
5  newtype T3 a = T3 (a -> a)
6  -- a is in both -ve and +ve position. T3 is invariant wrt. a.
7  newtype T4 a = T4 ((Int -> a) -> Int)
8  -- a is in +ve position but (Int -> a) is in -ve position.
9  -- So a is in -ve position overall. T4 is contravariant wrt. a.
10 newtype T5 a = T5 ((a -> Int) -> Int)
11 -- a is in -ve position but (a -> Int) is in -ve position.
12 -- So a is in +ve position overall. T5 is covariant wrt. a.

```

- Covariant parameters are said to be *produced* or *owned* by the type.
- Contravariant parameters are said to be *consumed* by the type.
- A type that has two parameters and is covariant in both of them is an instance of `BiFunctor`.
- A type that has two parameters and is contravariant in first parameter and covariant in second parameter is an instance of `Profunctor`.

Chapter 4. Working with Types

- Standard Haskell has no notion of scopes for types.
- `-XScopedTypeVariables` extension lets us bind type variables to a scope. It requires an explicitly `forall` quantifier in type signatures.

```

1  -- This does not compile.
2  > :{
3  | comp :: (a -> b) -> (b -> c) -> a -> c
4  | comp f g a = go f
5  |   where
6  |     go :: (a -> b) -> c
7  |     go f' = g (f' a)
8  | :}
9
10 <interactive>:11:11: error:
11   • Couldn't match expected type 'c1' with actual type 'c'
12     'c1' is a rigid type variable bound by
13       the type signature for:
14         go :: forall a1 b1 c1. (a1 -> b1) -> c1
15         at <interactive>:10:3-21
16     'c' is a rigid type variable bound by
17       the type signature for:
18         comp :: forall a b c. (a -> b) -> (b -> c) -> a -> c
19         at <interactive>:7:1-38
20   • In the expression: g (f' a)
21
22 <interactive>:11:14: error:
23   • Couldn't match expected type 'b' with actual type 'b1'
24     'b1' is a rigid type variable bound by
25       the type signature for:
26         go :: forall a1 b1 c1. (a1 -> b1) -> c1
27         at <interactive>:10:3-21
28     'b' is a rigid type variable bound by
29       the type signature for:
30         comp :: forall a b c. (a -> b) -> (b -> c) -> a -> c
31         at <interactive>:7:1-38
32   • In the first argument of 'g', namely '(f' a)'
33
34 <interactive>:11:17: error:
35   • Couldn't match expected type 'a1' with actual type 'a'
36     'a1' is a rigid type variable bound by
37       the type signature for:
38         go :: forall a1 b1 c1. (a1 -> b1) -> c1
39         at <interactive>:10:3-21
40     'a' is a rigid type variable bound by
41       the type signature for:
42         comp :: forall a b c. (a -> b) -> (b -> c) -> a -> c
43         at <interactive>:7:1-38
44   • In the first argument of 'f'', namely 'a'
45
46 -- But this does.
47 > :set -XScopedTypeVariables
48 > :{
49 | comp :: forall a b c. (a -> b) -> (b -> c) -> a -> c
50 | comp f g a = go f

```

```

51 | where
52 |   go :: (a -> b) -> c
53 |   go f' = g (f' a)
54 | :}

```

- -XTypeApplications extension lets us directly apply types to expressions:

```

1  > :set -XTypeApplications
2  > :type traverse
3  traverse
4  :: (Traversable t, Applicative f) =>
5      (a -> f b) -> t a -> f (t b)
6  > :type traverse @Maybe
7  traverse @Maybe
8  :: Applicative f =>
9      (a -> f b) -> Maybe a -> f (Maybe b)
10 > :type traverse @Maybe @[]
11 traverse @Maybe @[]
12 :: (a -> [b]) -> Maybe a -> [Maybe b]
13 > :type traverse @Maybe @[] @Int
14 traverse @Maybe @[] @Int
15 :: (Int -> [b]) -> Maybe Int -> [Maybe b]
16 > :type traverse @Maybe @[] @Int @String
17 traverse @Maybe @[] @Int @String
18 :: (Int -> [String]) -> Maybe Int -> [Maybe String]

```

- Types are applied in the order they appear in the type signature. It is possible to avoid applying types by using a type with an underscore: @_

```

1 > :type traverse @Maybe @_ @_ @String
2 traverse @Maybe @_ @_ @String
3 :: Applicative w1 =>
4     (w2 -> w1 String) -> Maybe w2 -> w1 (Maybe String)

```

- Sometimes the compiler cannot infer the type of an expression. -XAllowAmbiguousTypes extension allow such programs to compile.

```

1 > :set -XScopedTypeVariables
2 > :{
3 | f :: forall a. Show a => Bool
4 | f = True
5 | :}
6
7 <interactive>:7:6: error:
8   • Could not deduce (Show a0)
9     from the context: Show a
10    bound by the type signature for:
11        f :: forall a. Show a => Bool
12    at <interactive>:7:6-29
13    The type variable 'a0' is ambiguous
14   • In the ambiguity check for 'f'
15     To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
16     In the type signature: f :: forall a. Show a => Bool

```

- ▶ Proxy is a type isomorphic to () except with a phantom type parameter:

```

1 data Proxy a = Proxy

```

- ▶ With all the three extensions enabled, it is possible to get a term-level representation of types using the Data.Typeable module:

```

1 > :set -XScopedTypeVariables
2 > :set -XTypeApplications
3 > :set -XAllowAmbiguousTypes
4 > :m +Data.Typeable
5 Data.Typeable> :{
6 Data.Typeable| typeName :: forall a. Typeable a => String
7 Data.Typeable| typeName = show . typeRep $ Proxy @a
8 Data.Typeable| :}
9 Data.Typeable> typeName @String
10 "[Char]"
11 Data.Typeable> typeName @(IO Int)
12 "IO Int"

```

Chapter 5. Constraints and GADTs

Constraints

- ▶ *Constraints* are a kind different than the types (*).
- ▶ Constraints are what appear on the left-hand side on the fat context arrow =>, like Show a.

```

1 > :k Show
2 Show :: * -> Constraint
3 > :k Show Int
4 Show Int :: Constraint
5 > :k (Show Int, Eq String)
6 (Show Int, Eq String) :: Constraint

```

- ▶ Type equalities ($\text{Int} \sim a$) are another way of creating Constraints. ($\text{Int} \sim a$) says a is same as Int .
- ▶ Type equalities are
 - ▶ reflexive: $a \sim a$ always
 - ▶ symmetrical: $a \sim b$ implies $b \sim a$
 - ▶ transitive: $a \sim b$ and $b \sim c$ implies $a \sim c$

GADTs

- ▶ *GADTs* are Generalized Algebraic DataTypes. They allow writing explicit type signatures for data constructors. Here is the code for a length-typed list using GADTs:

```

1 > :set -XGADTs
2 > :set -XKindSignatures
3 > :set -XTypeOperators
4 > :set -XDataKinds
5 > :m +GHC.TypeLits
6 GHC.TypeLits> :{
7   GHC.TypeLits| data List (a :: *) (n :: Nat) where
8   GHC.TypeLits|   Nil  :: List a 0
9   GHC.TypeLits|   (:) :: a -> List a n -> List a (n + 1)
10  GHC.TypeLits| infixr 5 :~
11  GHC.TypeLits| :}
12 GHC.TypeLits> :type Nil
13 Nil :: List a 0
14 GHC.TypeLits> :type 'a' :~ Nil
15 'a' :~ Nil :: List Char 1
16 GHC.TypeLits> :type 'b' :~ 'a' :~ Nil
17 'b' :~ 'a' :~ Nil :: List Char 2
18 GHC.TypeLits> :type True :~ 'a' :~ Nil
19
20 <interactive>:1:9: error:
21   • Couldn't match type 'Char' with 'Bool'
22     Expected type: List Bool 1
23     Actual type: List Char (0 + 1)
24   • In the second argument of '(~)', namely 'a' :~ Nil'
25     In the expression: True :~ 'a' :~ Nil

```

- ▶ GADTs are just syntactic sugar for ADTs with type equalities. The above definition is equivalent to:

```

1  > :set -XGADTs
2  > :set -XKindSignatures
3  > :set -XTypeOperators
4  > :set -XDataKinds
5  > :m +GHC.TypeLits
6  GHC.TypeLits> :{
7  GHC.TypeLits| data List (a :: *) (n :: Nat)
8  GHC.TypeLits|   = (n ~ 0) => Nil
9  GHC.TypeLits|   | a :~ List a (n - 1)
10 GHC.TypeLits| infixr 5 :~
11 GHC.TypeLits| :}
12 GHC.TypeLits> :type 'a' :~ Nil
13 'a' :~ Nil :: List Char 1
14 GHC.TypeLits> :type 'b' :~ 'a' :~ Nil
15 'b' :~ 'a' :~ Nil :: List Char 2

```

- Type-safety of this list can be used to write a safe head function which does not compile for an empty list:

```

1  GHC.TypeLits> :{
2  GHC.TypeLits| safeHead :: List a (n + 1) -> a
3  GHC.TypeLits| safeHead (x :~ _) = x
4  GHC.TypeLits| :}
5  GHC.TypeLits> safeHead ('a' :~ 'b' :~ Nil)
6  'a'
7  GHC.TypeLits> safeHead Nil
8
9  <interactive>:21:10: error:
10      • Couldn't match type '1' with '0'
11          Expected type: List a (0 + 1)
12          Actual type: List a 0
13      • In the first argument of 'safeHead', namely 'Nil'
14          In the expression: safeHead Nil
15          In an equation for 'it': it = safeHead Nil

```

Heterogeneous Lists

We can use GADTs to build heterogeneous lists which can store values of different types and are type-safe to use.¹

First, the required extensions and imports:

```

1 {-# LANGUAGE KindSignatures #-}
2 {-# LANGUAGE DataKinds #-}
3 {-# LANGUAGE TypeOperators #-}
4 {-# LANGUAGE GADTs #-}
5 {-# LANGUAGE FlexibleInstances #-}
6 {-# LANGUAGE FlexibleContexts #-}
7 {-# LANGUAGE TypeApplications #-}
8 {-# LANGUAGE ScopedTypeVariables #-}
9
10 module HList where
11
12 import Data.Typeable

```

HList is defined as a GADT:

```

1 data HList (ts :: [*]) where
2   HNil :: HList '[]
3   (:#) :: t -> HList ts -> HList (t ': ts)
4 infixr 5 :#

```

Example usage:

```

1 *HList> :type HNil
2 HNil :: HList '[]
3 *HList> :type 'a' :# HNil
4 'a' :# HNil :: HList '[Char]
5 *HList> :type True :# 'a' :# HNil
6 True :# 'a' :# HNil :: HList '[Bool, Char]

```

We can write operations on HList:

```

1 hLength :: HList ts -> Int
2 hLength HNil = 0
3 hLength (x :# xs) = 1 + hLength xs
4
5 hHead :: HList (t ': ts) -> t
6 hHead (t :# _) = t

```

Example usage:

```

1  *HList> hLength $ True :# 'a' :# HNil
2  2
3  *HList> hHead $ True :# 'a' :# HNil
4  True
5  *HList> hHead HNil
6
7  <interactive>:7:7: error:
8      • Couldn't match type '['[]' with 't : ts0'
9        Expected type: HList (t : ts0)
10       Actual type: HList '['[]
11      • In the first argument of 'hHead', namely 'HNil'
12        In the expression: hHead HNil
13        In an equation for 'it': it = hHead HNil
14      • Relevant bindings include it :: t (bound at <interactive>:7:1)

```

We need to define instances of typeclasses like Eq, Ord etc. for HList because GHC cannot derive them automatically yet:

```

1  instance Eq (HList '['[])) where
2      HNil == HNil = True
3  instance (Eq t, Eq (HList ts))
4      => Eq (HList (t ':' ts)) where
5      (x :# xs) == (y :# ys) =
6          x == y && xs == ys
7
8  instance Ord (HList '['[])) where
9      HNil `compare` HNil = EQ
10 instance (Ord t, Ord (HList ts))
11     => Ord (HList (t ':' ts)) where
12     (x :# xs) `compare` (y :# ys) =
13         x `compare` y <> xs `compare` ys
14
15 instance Show (HList '['[])) where
16     show HNil = "[]"
17 instance (Typeable t, Show t, Show (HList ts))
18     => Show (HList (t ':' ts)) where
19     show (x :# xs) =
20         show x
21         ++ "@" ++ show (typeRep (Proxy @t))
22         ++ " :# " ++ show xs

```

The instances are defined recursively: one for the base case and one for the inductive case.

Example usage:


```

1  *HList> True :# 'a' :# HNil == True :# 'a' :# HNil
2  True
3  *HList> True :# 'a' :# HNil == True :# 'b' :# HNil
4  False
5  *HList> True :# 'a' :# HNil == True :# HNil
6
7  <interactive>:17:24: error:
8      • Couldn't match type '[']' with '['Char]
9        Expected type: HList '[Bool, Char]
10       Actual type: HList '[Bool]
11      • In the second argument of '(==)', namely 'True :# HNil'
12        In the expression: True :# 'a' :# HNil == True :# HNil
13        In an equation for 'it': it = True :# 'a' :# HNil == True :# HNil
14  *HList> show $ True :# 'a' :# HNil
15  "True@Bool :# 'a'@Char :# []"

```

Creating New Constraints

- ▶ Type families can be used to create new Constraints:

```

1  > :set -XKindSignatures
2  > :set -XDataKinds
3  > :set -XTypeOperators
4  > :set -XTypeFamilies
5  > :m +Data.Constraint
6  Data.Constraint> :{
7  Data.Constraint| type family AllEq (ts :: [*]) :: Constraint where
8  Data.Constraint|   AllEq '[] = ()
9  Data.Constraint|   AllEq (t ': ts) = (Eq t, AllEq ts)
10 Data.Constraint| :}
11 Data.Constraint> :kind! AllEq '[Bool, Char]
12 AllEq '[Bool, Char] :: Constraint
13 = (Eq Bool, (Eq Char, () :: Constraint))

```

- ▶ AllEq is a type-level function from a list of types to a constraint.
- ▶ With the -XConstraintKinds extension, AllEq can be made polymorphic over all constraints instead of just Eq:

```

1  > :set -XConstraintKinds
2  Data.Constraint> :{
3  Data.Constraint| type family All (c :: * -> Constraint)
4  Data.Constraint|   (ts :: [*]) :: Constraint where
5  Data.Constraint|   All c '[] = ()
6  Data.Constraint|   All c (t ': ts) = (c t, All c ts)
7  Data.Constraint| :}

```

- ▶ With All, instances for HList can be written non-recursively:

```
1 instance All Eq ts => Eq (HList ts) where
2   HNil == HNil = True
3   (a :# as) == (b :# bs) = a == b && as == bs
```

Conclusion

I'm still in the process of reading the book and I'll post the notes for the rest of the chapters in a later post. For now, you can discuss this post on [lobsters](#), [r/haskell](#), [hacker news](#), [twitter](#) or in the comments below.

Footnotes

1. The complete code for `HList`.↩