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

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.

```
data Void
 -- no possible values. cardinality: 0
data Unit = Unit
 -- only one possible value. cardinality: 1
data Bool = True | False
 -- 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:

```
to :: a -> b
from :: b -> a
to . from = id
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:

```
|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:

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

- Some more examples:

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

- Function types are exponentiation types.

```
|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:

```
|a -> b|
  = |b| * |b| * ... * |b| -- (|a| times)
  = |b|^|a|
```

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

```haskell
data TicTacToe1 a = TicTacToe1
  { topLeft      :: a
  , topCenter    :: a
  , topRight     :: a
  , middleLeft   :: a
  , middleCenter :: a
  , middleRight  :: a
  , bottomLeft   :: a
  , bottomCenter :: a
  , bottomRight  :: a
  }

|TicTacToe1 a|
  = |a| * |a| * ... * |a| -- 9 times
  = |a|^9

emptyBoard1 :: TicTacToe1 (Maybe Bool)
emptyBoard1 =
  TicTacToe1 Nothing Nothing Nothing
             Nothing Nothing Nothing
             Nothing Nothing Nothing

-- Alternatively

data Three = One | Two | Three
data TicTacToe2 a =
  TicTacToe2 (Three -> Three -> a)

|TicTacToe2 a| = |a|^(|Three| * |Three|)
               = |a|^(3*3)
               = |a|^9

emptyBoard2 :: TicTacToe2 (Maybe Bool)
emptyBoard2 =
  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:

```haskell
Either a (Either b (c, d)) -- canonical

(a, Bool) -- not canonical
Either a a
-- 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 `*`.

```haskell
> :type True
True :: Bool
> :kind Bool
Bool :: *
```

- There are kinds other than `*`. For example:

```haskell
> :kind Show Int
Show Int :: Constraint
```

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

```haskell
 > :kind Maybe
 Maybe :: * -> *
 > :kind Maybe Int
 Maybe Int :: *

 > :type Control.Monad.Trans.Maybe.MaybeT
 Control.Monad.Trans.Maybe.MaybeT
   :: m (Maybe a) -> Control.Monad.Trans.Maybe.MaybeT m a
 > :kind Control.Monad.Trans.Maybe.MaybeT
 Control.Monad.Trans.Maybe.MaybeT :: (* -> *) -> * -> *
 > :kind Control.Monad.Trans.Maybe.MaybeT IO Int
 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.

```
> :set -XDataKinds
> data Allow = Yes | No
> :type Yes
Yes :: Allow
-- Yes is data constructor
> :kind Allow -- Allow is a type
Allow :: *
> :kind 'Yes
'Yes :: Allow
-- '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`.

```
> :kind "hi"
"hi" :: GHC.Types.Symbol
-- "hi" is a type-level string
> :kind 123
123 :: GHC.Types.Nat
-- 123 is a type-level natural number
```

- We can do type level operations on `Symbol`s and `Nat`s.

```
> :m +GHC.TypeLits
GHC.TypeLits> :kind AppendSymbol
AppendSymbol :: Symbol -> Symbol -> Symbol
GHC.TypeLits> :kind! AppendSymbol "hello " "there"
AppendSymbol "hello " "there" :: Symbol
= "hello there"
GHC.TypeLits> :set -XTypeOperators
GHC.TypeLits> :kind! (1 + 2) ^ 7
(1 + 2) ^ 7 :: Nat
= 2187
```

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

```
GHC.TypeLits> :kind '[ 'True ]
'[ 'True ] :: [Bool]
GHC.TypeLits> :kind '[1,2,3]
'[1,2,3] :: [Nat]
GHC.TypeLits> :kind '["abc"]
'["abc"] :: [Symbol]
GHC.TypeLits> :kind 'False ': 'True ': '[]
'False ': 'True ': '[] :: [Bool]
GHC.TypeLits> :kind '(6, "x", 'False)
'(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:

```
> :set -XDataKinds
> :set -XTypeFamilies
> :{
| type family And (x :: Bool) (y :: Bool) :: Bool where
|   And 'True 'True = 'True
|   And _     _     = 'False
| :}
> :kind And
And :: Bool -> Bool -> Bool
> :kind! And 'True 'False
And 'True 'False :: Bool
= 'False
> :kind! And 'True 'True
And 'True 'True :: Bool
= 'True
> :kind! And 'False 'True
And 'False 'True :: Bool
= 'False
```

Chapter 3. Variance

- There are three types of *Variance* (`T` here a type of kind `* -> *`):
  - Covariant: any function of type `a -> b` can be lifted into a function of type `T a -> T b`. Covariant types are instances of the `Functor` typeclass:

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

  - Contravariant: any function of type `a -> b` can be lifted into a function of type `T b -> T a`. Contravariant functions are instances of the `Contravariant` typeclass:

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

  - Invariant: no function of type `a -> b` can be lifted into a function of type `T a`. Invariant functions are instances of the `Invariant` typeclass:

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

- Variance of a type `T` is specified with respect to a particular type parameter. A type `T` with two parameters `a` and `b` could be covariant wrt. `a` and contravariant wrt. `b`.
- Variance of a type `T` wrt. a particular type parameter is determined by whether the parameter appears in positive or negative *position* s.
  - 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 *sign* s.

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

- Examples:

```haskell
newtype T1 a = T1 (Int -> a)
-- a is in +ve position, T1 is covariant wrt. a.
newtype T2 a = T2 (a -> Int)
-- a is in -ve position, T2 is contravariant wrt. a.
newtype T3 a = T3 (a -> a)
-- a is in both -ve and +ve position. T3 is invariant wrt. a.
newtype T4 a = T4 ((Int -> a) -> Int)
-- a is in +ve position but (Int -> a) is in -ve position.
-- So a is in -ve position overall. T4 is contravariant wrt. a.
newtype T5 a = T5 ((a -> Int) -> Int)
-- a is in -ve position but (a -> Int) is in -ve position.
-- 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.

```
-- This does not compile.
> :{
| comp :: (a -> b) -> (b -> c) -> a -> c
| comp f g a = go f
|  where
|   go :: (a -> b) -> c
|   go f' = g (f' a)
| :}

<interactive>:11:11: error:
    • Couldn't match expected type ‘c1’ with actual type ‘c’
      ‘c1’ is a rigid type variable bound by
        the type signature for:
          go :: forall a1 b1 c1. (a1 -> b1) -> c1
        at <interactive>:10:3-21
      ‘c’ is a rigid type variable bound by
        the type signature for:
          comp :: forall a b c. (a -> b) -> (b -> c) -> a -> c
        at <interactive>:7:1-38
    • In the expression: g (f' a)

<interactive>:11:14: error:
    • Couldn't match expected type ‘b’ with actual type ‘b1’
      ‘b1’ is a rigid type variable bound by
        the type signature for:
          go :: forall a1 b1 c1. (a1 -> b1) -> c1
        at <interactive>:10:3-21
      ‘b’ is a rigid type variable bound by
        the type signature for:
          comp :: forall a b c. (a -> b) -> (b -> c) -> a -> c
        at <interactive>:7:1-38
    • In the first argument of ‘g’, namely ‘(f' a)’

<interactive>:11:17: error:
    • Couldn't match expected type ‘a1’ with actual type ‘a’
      ‘a1’ is a rigid type variable bound by
        the type signature for:
          go :: forall a1 b1 c1. (a1 -> b1) -> c1
        at <interactive>:10:3-21
      ‘a’ is a rigid type variable bound by
        the type signature for:
          comp :: forall a b c. (a -> b) -> (b -> c) -> a -> c
        at <interactive>:7:1-38
    • In the first argument of ‘f'’, namely ‘a’

-- But this does.
> :set -XScopedTypeVariables
> :{
| comp :: forall a b c. (a -> b) -> (b -> c) -> a -> c
| comp f g a = go f
```

```
|  where
|   go :: (a -> b) -> c
|   go f' = g (f' a)
| :}
```

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

```
> :set -XTypeApplications
> :type traverse
traverse
  :: (Traversable t, Applicative f) =>
     (a -> f b) -> t a -> f (t b)
> :type traverse @Maybe
traverse @Maybe
  :: Applicative f =>
     (a -> f b) -> Maybe a -> f (Maybe b)
> :type traverse @Maybe @[]
traverse @Maybe @[]
  :: (a -> [b]) -> Maybe a -> [Maybe b]
> :type traverse @Maybe @[] @Int
traverse @Maybe @[] @Int
  :: (Int -> [b]) -> Maybe Int -> [Maybe b]
> :type traverse @Maybe @[] @Int @String
traverse @Maybe @[] @Int @String
  :: (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: `@_`

```
> :type traverse @Maybe @_ @_ @String
traverse @Maybe @_ @_ @String
  :: Applicative w1 =>
     (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.

```
> :set -XScopedTypeVariables
> :{
| f :: forall a. Show a => Bool
| f = True
| :}

<interactive>:7:6: error:
    • Could not deduce (Show a0)
      from the context: Show a
        bound by the type signature for:
                   f :: forall a. Show a => Bool
        at <interactive>:7:6-29
      The type variable ‘a0’ is ambiguous
    • In the ambiguity check for ‘f’
      To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
      In the type signature: f :: forall a. Show a => Bool
```

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

```
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:

```
> :set -XScopedTypeVariables
> :set -XTypeApplications
> :set -XAllowAmbiguousTypes
> :m +Data.Typeable
Data.Typeable> :{
Data.Typeable| typeName :: forall a. Typeable a => String
Data.Typeable| typeName = show . typeRep $ Proxy @a
Data.Typeable| :}
Data.Typeable> typeName @String
"[Char]"
Data.Typeable> typeName @(IO Int)
"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`.

```
> :k Show
Show :: * -> Constraint
> :k Show Int
Show Int :: Constraint
> :k (Show Int, Eq String)
(Show Int, Eq String) :: Constraint
```

- Type equalities `(Int ~ a)` are another way of creating Constraints. `(Int ~ a)` says `a` is same as `Int`.
- Type equalities are
  - reflexive: `a ~ a` always
  - symmetrical: `a ~ b` implies `b ~ a`
  - transitive: `a ~ b` and `b ~ c` implies `a ~ 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:

```
> :set -XGADTs
> :set -XKindSignatures
> :set -XTypeOperators
> :set -XDataKinds
> :m +GHC.TypeLits
GHC.TypeLits> :{
GHC.TypeLits| data List (a :: *) (n :: Nat) where
GHC.TypeLits|   Nil  :: List a 0
GHC.TypeLits|   (:~) :: a -> List a n -> List a (n + 1)
GHC.TypeLits| infixr 5 :~
GHC.TypeLits| :}
GHC.TypeLits> :type Nil
Nil :: List a 0
GHC.TypeLits> :type 'a' :~ Nil
'a' :~ Nil :: List Char 1
GHC.TypeLits> :type 'b' :~ 'a' :~ Nil
'b' :~ 'a' :~ Nil :: List Char 2
GHC.TypeLits> :type True :~ 'a' :~ Nil

<interactive>:1:9: error:
    • Couldn't match type ‘Char’ with ‘Bool’
      Expected type: List Bool 1
        Actual type: List Char (0 + 1)
    • In the second argument of ‘(:~)’, namely ‘'a' :~ Nil’
      In the expression: True :~ 'a' :~ Nil
```

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

```
> :set -XGADTs
> :set -XKindSignatures
> :set -XTypeOperators
> :set -XDataKinds
> :m +GHC.TypeLits
GHC.TypeLits> :{
GHC.TypeLits| data List (a :: *) (n :: Nat)
GHC.TypeLits|   = (n ~ 0) => Nil
GHC.TypeLits|   | a :~ List a (n - 1)
GHC.TypeLits| infixr 5 :~
GHC.TypeLits| :}
GHC.TypeLits> :type 'a' :~ Nil
'a' :~ Nil :: List Char 1
GHC.TypeLits> :type 'b' :~ 'a' :~ Nil
'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:

```
GHC.TypeLits> :{
GHC.TypeLits| safeHead :: List a (n + 1) -> a
GHC.TypeLits| safeHead (x :~ _) = x
GHC.TypeLits| :}
GHC.TypeLits> safeHead ('a' :~ 'b' :~ Nil)
'a'
GHC.TypeLits> safeHead Nil

<interactive>:21:10: error:
    • Couldn't match type ‘1’ with ‘0’
      Expected type: List a (0 + 1)
        Actual type: List a 0
    • In the first argument of ‘safeHead’, namely ‘Nil’
      In the expression: safeHead Nil
      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:

```haskell
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE ScopedTypeVariables #-}

module HList where

import Data.Typeable
```

`HList` is defined as a GADT:

```haskell
data HList (ts :: [*]) where
  HNil :: HList '[]
  (:#) :: t -> HList ts -> HList (t ': ts)
infixr 5 :#
```

Example usage:

```haskell
*HList> :type HNil
HNil :: HList '[]
*HList> :type 'a' :# HNil
'a' :# HNil :: HList '[Char]
*HList> :type True :# 'a' :# HNil
True :# 'a' :# HNil :: HList '[Bool, Char]
```

We can write operations on `HList`:

```haskell
hLength :: HList ts -> Int
hLength HNil = 0
hLength (x :# xs) = 1 + hLength xs

hHead :: HList (t ': ts) -> t
hHead (t :# _) = t
```

Example usage:

```
*HList> hLength $ True :# 'a' :# HNil
2
*HList> hHead $ True :# 'a' :# HNil
True
*HList> hHead HNil

<interactive>:7:7: error:
    • Couldn't match type ‘'[]’ with ‘t : ts0’
      Expected type: HList (t : ts0)
        Actual type: HList '[]
    • In the first argument of ‘hHead’, namely ‘HNil’
      In the expression: hHead HNil
      In an equation for ‘it’: it = hHead HNil
    • 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:

```
instance Eq (HList '[]) where
  HNil == HNil = True
instance (Eq t, Eq (HList ts))
    => Eq (HList (t ': ts)) where
  (x :# xs) == (y :# ys) =
    x == y && xs == ys

instance Ord (HList '[]) where
  HNil `compare` HNil = EQ
instance (Ord t, Ord (HList ts))
    => Ord (HList (t ': ts)) where
  (x :# xs) `compare` (y :# ys) =
    x `compare` y <> xs `compare` ys

instance Show (HList '[]) where
  show HNil = "[]"
instance (Typeable t, Show t, Show (HList ts))
    => Show (HList (t ': ts)) where
  show (x :# xs) =
    show x
    ++ "@" ++ show (typeRep (Proxy @t))
    ++ " :# " ++ show xs
```

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

Example usage:

```
*HList> True :# 'a' :# HNil == True :# 'a' :# HNil
True
*HList> True :# 'a' :# HNil == True :# 'b' :# HNil
False
*HList> True :# 'a' :# HNil == True :# HNil

<interactive>:17:24: error:
    • Couldn't match type ‘'[]’ with ‘'[Char]’
      Expected type: HList '[Bool, Char]
        Actual type: HList '[Bool]
    • In the second argument of ‘(==)’, namely ‘True :# HNil’
      In the expression: True :# 'a' :# HNil == True :# HNil
      In an equation for ‘it’: it = True :# 'a' :# HNil == True :# HNil
*HList> show $ True :# 'a' :# HNil
"True@Bool :# 'a'@Char :# []"
```

Creating New Constraints

- Type families can be used to create new Constraints:

```
> :set -XKindSignatures
> :set -XDataKinds
> :set -XTypeOperators
> :set -XTypeFamilies
> :m +Data.Constraint
Data.Constraint> :{
Data.Constraint| type family AllEq (ts :: [*]) :: Constraint where
Data.Constraint|   AllEq '[] = ()
Data.Constraint|   AllEq (t ': ts) = (Eq t, AllEq ts)
Data.Constraint| :}
Data.Constraint> :kind! AllEq '[Bool, Char]
AllEq '[Bool, Char] :: Constraint
= (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`:

```
> :set -XConstraintKinds
Data.Constraint> :{
Data.Constraint| type family All (c :: * -> Constraint)
Data.Constraint|                  (ts :: [*]) :: Constraint where
Data.Constraint|   All c '[] = ()
Data.Constraint|   All c (t ': ts) = (c t, All c ts)
Data.Constraint| :}
```

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

```haskell
instance All Eq ts => Eq (HList ts) where
  HNil == HNil = True
  (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 or in the comments below.

Footnotes

1. The complete code for `HList`.↩