The Monad.Reader Mini Issue 24: Predicates, Trees and GADTs

by Benjamin Hodgson ⟨benhodgson91@gmail.com⟩

August 14, 2015

Edward Z. Yang, editor.

Predicates, Trees and GADTs

by Benjamin Hodgson ⟨benjamin.hodgson@huddle.com⟩

Haskell's crazy type system marches boldly past polymorphism and into the no-man's land between us and dependent types. But GHC's advanced features are not just a hobby-horse for researchers. In this article, we describe an application of constraints, higher-ranked types, GADTs, and type-level data to a problem which we encountered in a real-world setting, namely the definition of a ***predicate language*** *for security checks.*

Introducing Predicates

Where I work [1], we've built a datatype of composable Boolean predicates to help us implement security checks. It works by combining small functions $a \to Bool$ into a tree which admits many interpretations: you can evaluate trees back into a function, pretty-print them, compare them for equality, and so on. My colleagues and I are stuck using C♯, but the idea translates into Haskell concisely and beautifully:

```
type Name = String
data Pred a = Leaf Name (a → Bool)
  | And (Pred a) (Pred a)
  | Or (Pred a) (Pred a)
  | Not (Pred a)
```

A *Pred a* denotes a predicate on some type *a*; soon we will write a function *eval* with a type of $Pred\ a \to (a \to Bool)$ which pulls out the function which a *Pred* is meant to represent. At the leaves of the tree are small functions which test a single fact about *a*, along with a name for the leaf. The other constructors provide ways to compose predicates together: *And* denotes the conjunction of two predicates, *Or* denotes a disjunction, and *Not* denotes the negation of a predicate. (There is some overlap between these denotations; *Or p q* is equivalent to

Not (*And* (*Not* *p*) (*Not* *q*)). I chose to keep the redundant constructors to make the code clearer.)

Why go to the effort of developing a whole data type for predicates? We could represent predicates as functions $a \to Bool$ and provide combinators which compose them. But a function is a black box; all you can do with a function is apply it to something. Reifying the *And*, *Or* and *Not* combinations as a data type allows you to manipulate and analyse the structure of a predicate – for example, you can compare two *Pred*s for equality – whereas functions admit no such manipulation.

Here's how you might use *Pred* for permissions checks in a simple blogging system. First, the players on our stage:

```
data User = U {
  username :: String,
  isRegistered :: Bool
} deriving (Eq)
data Post = P {
  isPublic :: Bool,
  canCommentAnonymously :: Bool,
  author :: User
}
```

How might we write a *Pred* to check whether a user is allowed to edit a given post? We'll need to query information about both the user and the post, but *Pred* has only one type parameter. So we simply wrap up our two entities into a new type representing the context of an action within the blogging system: the user performing the action and the post they are acting upon.

```
data World = W {
  user :: User,
  post :: Post
}
```

Here are a bunch of permissions checks, all with a type of *Pred World*. The *And*, *Or* and *Not* constructors naturally form a simple domain-specific language, thanks to Haskell's support for infix notation.

```
userIsRegistered = Leaf "userIsRegistered" (isRegistered ∘ user)
postIsPublic = Leaf "postIsPublic" (isPublic ∘ post)
userIsAuthor = Leaf "userIsAuthor" (λx → author (post x) ≡ user x)
postAllowsAnonComments =
  Leaf "postAllowsAnonComments" (canCommentAnonymously ∘ post)
userCanComment = (userIsRegistered ‘Or‘ postAllowsAnonComments)
```

```
    `And` postIsPublic
userCanEditPost = userIsAuthor `And` userIsRegistered
```

Higher-order functions

How should we implement *eval* :: *Pred a* → *a* → *Bool*? We could use recursion...

```
eval (Leaf name f) x = f x
eval (And l r) x = eval l x ∧ eval r x
eval (Or l r) x = eval l x ∨ eval r x
eval (Not p) x = ¬ (eval p x)
```

...but as every functional programmer knows, it's usually better to provide higher-order functions to process your data structures. So here's a useful, general function for reducing a *Pred* to a value.

```
fold :: (Name → (a → Bool) → b) →    -- map the leaves
   (b → b → b) →                     -- combine And nodes
   (b → b → b) →                     -- combine Or nodes
   (b → b) →                         -- transform Not nodes
   Pred a → b
fold leaf and or not (Leaf name f) = leaf name f
fold leaf and or not (And l r) =
   (fold leaf and or not l) `and` (fold leaf and or not r)
fold leaf and or not (Or l r) =
   (fold leaf and or not l) `or` (fold leaf and or not r)
fold leaf and or not (Not p) = not (fold leaf and or not p)
```

Look how beautiful *eval* is when we implement it in terms of *fold*!

```
eval :: Pred a → a → Bool
eval p x = fold (λname f → f x) (∧) (∨) (¬) p
```

We can also do other things using *fold*, such as printing out a human-readable representation of a *Pred*.

```
prettyPrint :: Pred a → String
prettyPrint = fold ppLeaf ppAnd ppOr ppNot
   where ppLeaf name f = name
      ppAnd l r = "(" ++ l ++ " AND " ++ r ++ ")"
      ppOr l r = "(" ++ l ++ " OR " ++ r ++ ")"
      ppNot x = "(NOT " ++ x ++ ")"
```

It's always fun to implement the identity function in terms of a fold:

```
predId :: Pred a → Pred a
predId = fold Leaf And Or Not
```

Composability

Go back and look at the blogging example again. That *World* type is a bit of a pain. It arose because we wanted to write predicates which operate on more than one entity, but you can't directly compose a *Pred Post* with a *Pred User*. *World* doesn't really have a semantic meaning; it's simply a place to put "data you might want to query for a permission check". The problem is, if we wanted to check permissions on a new entity, we'd have to change *World* and break backwards compatibility.

```
data Comment = C {
  commentAuthor :: User,
  commentText :: String
}
userCanDeleteComment = Leaf "userCanDeleteComment" (λx →
  commentAuthor (comment x) ≡ user x)

data World = W {
  user :: User,
  post :: Post,
  comment :: Comment   -- we had to add this
}
```

There'll now be places where the *W* constructor is being used but a *Comment* is not available, forcing people who want to use our predicates to set that field to ⊥ and risk crashing their program! This might make sense in a dynamically typed language, but it seems like the wrong approach in Haskell. The problem gets worse as we add more entities to our system: the bigger the *World* becomes, the harder it is to use. *World* does a poor job of solving the problem of composing predicates of different types.

The root cause of the pain is that predicates are too restrictive about the type of data that they test. *userIsActive* only needs a *User*, but its type (*Pred World*) insists on dragging a whole *World* along with it. This results in an implicit coupling between the particular predicate we use and the contents of the *World* you use it with.

Type classes allow us to make that implicit coupling explicit in the type. I'm going to rewrite our atomic predicates so that they don't specify a particular type; they just say what they need from a candidate. The goal is to defer the choice of type *a* to the use site of *eval*.

```
class HasUser a where
  user :: a → User
class HasPost a where
  post :: a → Post
class HasComment a where
  comment :: a → Comment
```

The type class methods *user*, *post* and *comment* can be used as drop-in replacements for the extractors that were previously defined as fields in *World*. Our original predicates compile without modification with more general types:

```
ghci> :t userIsRegistered
  userIsRegistered :: HasUser a => Pred a
ghci> :t userCanComment
  userCanComment :: (HasUser a, HasPost a) => Pred a
```

userCanComment was a composed predicate. Consequently, GHC has pushed the *HasUser* and *HasPost* requirements up the tree for us: you have to satisfy all the leaf nodes' constraints in order to use a predicate tree.

As I mentioned, this design has the effect of deferring the choice of type *a* to the place where the predicate is used. When you want to *eval* one of our new type class-based predicates, you make up a record type containing the required data and write instances of *HasUser*, *HasPost* and *HasComment* as appropriate. If your predicate doesn't need a *Comment* you aren't forced to provide one.

```
instance HasUser User where
  user = id
test1 = eval userIsRegistered (U "lambda" True)  -- True
data UserAndPost = UP User Post
instance HasUser UserAndPost where
  user (UP u p) = u
instance HasPost UserAndPost where
  post (UP u p) = p
test2 = eval userCanComment (UP u p)  -- False
  where
    u = U "forall" False
```

```
p = P {
  isPublic = True,
  canCommentAnonymously = False,
  author = U "lambda" True
}
```

Strengthening the API

What's problematic about this type class-based solution? It requires everyone to agree on the idea of using type classes instead of concrete types. Predicates have to be universally quantified, otherwise they don't compose any more. If I hastily write a *Pred MyCoolType* and compose it with some of your carefully designed predicates, the whole tree becomes a *Pred MyCoolType*, with all the same problems as the original version. The situation we wanted to avoid hasn't been ruled out, merely swept under the carpet; one uncooperative party can spoil the fun for everyone. Can *Pred* be re-engineered so that it may never be used with a specific type?

We want to keep the compositional properties of the type class-based design, while "baking in" the requirement of universally quantifying *a*. The plan is to parameterise *Pred* not by any exact type but by the type class constraint the predicate places on the type that it tests. *Pred User* will no longer be valid; instead we'll be writing types like *Pred HasUser*.

```
data Pred′ c where    -- Pred′ :: (∗ → Constraint) → ∗
  Leaf′ :: Name → (∀ a. c a ⇒ a → Bool) → Pred′ c
  And′ :: Pred′ c1 → Pred′ c2 → Pred′ (c1 ∧ c2)
  Or′ :: Pred′ c1 → Pred′ c2 → Pred′ (c1 ∧ c2)
  Not′ :: Pred′ c → Pred′ c
```

Look carefully at the types of these constructors. *Leaf′* has a rank-two type: the ∀ *a* means that the function in parentheses must be polymorphic in *a*. The author of a predicate is not free to choose a type *a* because we have hidden it from the *Pred* type – all you can say is that the concrete type with which the predicate is eventually used must satisfy a certain constraint *c*.

And′ and *Or′* have special types too: they join the constraints of their subtrees together into a single, stronger constraint. (∧, which we will define below, denotes the conjunction of two constraints.) The *Pred′* that you get back from an *And′* or *Or′* constructor has a stronger constraint than its two inputs because you have to be able to satisfy the constraints of both subtrees. In other words, We are manually propagating constraints from the leaves to the root of the tree, which GHC previously did implicitly.

I had to turn on some language extensions to write this data type. The `RankNTypes` extension enables **higher-rank types**, allowing us to ensure that the functions at the leaves of our tree are polymorphic; `ConstraintKinds` lets type class constraints such as *HasUser* roam free across types like any other type variable; and GADTs – **generalised algebraic data types** – mean the constructors of a type can have non-uniform return types.

We can explain how to push constraints together using type class Prolog. The joint constraint $(c1 \wedge c2)\ a$ can be discharged if and only if $c1\ a$ and $c2\ a$ hold separately.

```
class (c1 a, c2 a) ⇒ (∧) c1 c2 a
instance (c1 a, c2 a) ⇒ (∧) c1 c2 a
```

I'm going to unify the *Has*... classes from earlier into a single generic class.

```
  -- HasUser is equivalent to Get User and so on
class Get r a where
  get :: a → r
```

The blogging permission checks themselves look the same as they did before – I just adjusted them to use *Get*. Unfortunately, *Leaf′*'s second-rank type means that we lose type inference for the atomic predicates at the leaves of the tree.

```
userIsRegistered′ :: Pred′ (Get User)
userIsRegistered′ = Leaf′ "userIsRegistered’" (isRegistered ∘ get)
userIsAuthor′ :: Pred′ (Get Post ∧ Get User)
userIsAuthor′ = Leaf′ "userIsAuthor’" (λx → author (get x) ≡ get x)
userCanEditPost′ = userIsAuthor′ ‘And′‘ userIsRegistered′
```

Folding up a GADT

As before, we want to avoid resorting to primitive recursion when processing a *Pred′*, so let's think about the type of *fold* for this new version.

There are two requirements – which in fact apply whenever you need to fold up a GADT – that the type of *fold′* has to fulfil. The first is that we don't want folding up a predicate to erase our knowledge about the constraint c. (If c were not present in the return type of a fold, it'd be impossible to write $eval' :: Pred'\ c \to (c\ a \Rightarrow a \to Bool)$ as a fold.) This suggests that the return type of *fold′* should be some type f which is parameterised over a constraint: $fold' :: ... \to Pred'\ c \to f\ c$.

The second requirement arises due to the fact that *Pred′* is a GADT; the type parameter *c* changes as you walk around the tree. The functions you plug into *fold′* must therefore work over all constraints *c* – in other words, the folding functions will feature a ∀ *c* forcing them to be polymorphic.

```
type f ⇝ g = ∀ a. f a → g a
fold′ ::
  (∀ c. String → (∀ a. c a ⇒ a → Bool) → f c) →     -- map the leaves
  (∀ c1 c2. f c1 → f c2 → f (c1 ∧ c2)) →             -- combine And nodes
  (∀ c1 c2. f c1 → f c2 → f (c1 ∧ c2)) →             -- combine Or nodes
  (∀ c. f c → f c) →                                   -- transform Not nodes
  Pred′ ⇝ f
fold′ leaf and or not (Leaf′ name f) = leaf name f
fold′ leaf and or not (And′ l r) =
  (fold′ leaf and or not l) ‘and‘ (fold′ leaf and or not r)
fold′ leaf and or not (Or′ l r) =
  (fold′ leaf and or not l) ‘or‘ (fold′ leaf and or not r)
fold′ leaf and or not (Not′ p) = not (fold′ leaf and or not r)
```

That's a rank-three type up there! *fold′* is parameterised by a function polymorphic in *c* which is parameterised by a function polymorphic in *a*... my head is spinning. Higher-rank types can get confusing if they're over-used; they also break type inference, which means that users of *fold′* will have to give explicit type signatures for all the functions they want to plug in.

It turns out that type classes let us design a simpler API which satisfies the same requirements. Essentially, the higher-rank function parameters are translated into (overloaded) top-level functions. (Alternatively, you could put the folding functions in a record type and pass it around explicitly, effectively writing by hand the code that GHC will generate from a class.)

```
class PredFold f where
  foldLeaf :: Name → (∀ a. c a ⇒ a → Bool) → f c
  foldAnd :: f c1 → f c2 → f (c1 ∧ c2)
  foldOr :: f c1 → f c2 → f (c1 ∧ c2)
  foldNot :: f c → f c
fold′ :: PredFold f ⇒ Pred′ ⇝ f
fold′ (Leaf′ name f) = foldLeaf name f
fold′ (And′ l r) = foldAnd (fold′ l) (fold′ r)
fold′ (Or′ l r) = foldOr (fold′ l) (fold′ r)
fold′ (Not′ x) = foldNot (fold′ x)
```

Using *fold′* now consists of picking a type *f* :: (∗ → *Constraint*) → ∗ to fold the *Pred′* up into, and declaring an instance of *PredFold* containing the folding functions.

```
newtype Eval c = Eval {getEval :: c a ⇒ a → Bool}
instance PredFold Eval where
  foldLeaf name f = Eval f
  foldAnd (Eval l) (Eval r) = Eval $ λx → l x ∧ r x
  foldOr (Eval l) (Eval r) = Eval $ λx → l x ∨ r x
  foldNot (Eval p) = Eval $ λx → ¬ (p x)
eval′ :: c a ⇒ Pred′ c → a → Bool
eval′ = getEval ∘ fold′

newtype Const a b = Const {getConst :: a}
instance PredFold (Const String) where
  foldLeaf name f = Const name
  foldAnd (Const l) (Const r) = Const $ "(" ⧺ l ⧺ " AND " ⧺ r ⧺ ")"
  foldOr (Const l) (Const r) = Const $ "(" ⧺ l ⧺ " OR " ⧺ r ⧺ ")"
  foldNot (Const x) = Const $ "(NOT " ⧺ x ⧺ ")"
prettyPrint′ :: Pred′ c → String
prettyPrint′ = getConst ∘ fold′

instance PredFold Pred′ where
  foldLeaf = Leaf′
  foldAnd = And′
  foldOr = Or′
  foldNot = Not′
predId′ :: Pred′ c → Pred′ c
predId′ = fold′
```

(There's nothing stopping you from providing the higher-order-function version of *fold′* as well as the type class version. The difference is akin to *minimum* versus *minimumBy* in the Prelude.)

Generic tuples

Deferring the choice of type for the predicate to the use-site has real engineering advantages, but it also increases the boilerplate required to use a predicate. It's pretty tedious for users to write a boring record containing boring entities and a boring set of instances of *Get* every time they use a predicate. You could write some Template Haskell helpers to generate instances automatically, but it'd be more satisfying to define a general datatype which has an appropriate instance of *Get* by default. Let's develop a heterogeneous tuple whose *Get* instance picks out the first element of the appropriate type.

`DataKinds` has the effect of introducing **type-level data**: types can feature numbers, booleans, lists of other types, and so on. It works by duplicating all **data** declarations at the type level – **data** *Bool* = *True* | *False* now also introduces a kind called *Bool* and two types called *True* and *False*.

Type-level data gets useful when you use a GADT to glue values and types together. We're going to parameterise our tuple type by a list containing the types of the elements inside it.

```
infixr 5 :>
data Tuple as where
  E :: Tuple '[]
  (:>) :: a → Tuple as → Tuple (a ': as)
```

Tuples work rather like a (heterogeneous) linked list, with some extra type-level information about the things inside it. A tuple can be empty (E), in which case its associated list of types is also empty, or it can be a cons cell (:>) containing an element and the rest of the tuple. When you add something to the front of the tuple, you add its type to the front of the list of types, too. The tuple and its type grow together, like a vine creeping up a bamboo plant.

Let's query the type of a few tuples, to get a feel for how they work.

```
ghci> :t 'x' :> E
  'x' :> E :: Tuple '[Char]
ghci> :t [] :> 'x' :> E
  [] :> 'x' :> E :: Tuple '[[t], Char]
ghci> :t "pi" :> 'c' :> True :> E
  "pi" :> 'c' :> True :> E :: Tuple '[[Char], Char, Bool]
```

Now all that remains is to write an instance of *Get* for tuples. We want to pick out the first element which matches the type you're trying to get. We'd like to write this:

```
instance Get a (Tuple (a ': as))
  where get (x :> xs) = x
instance Get a (Tuple as) ⇒ Get a (Tuple (b ': as))
  where get (x :> xs) = get xs
```

but unfortunately it falls foul of GHC's overlapping instances rule when you use it.

```
ghci> get ("hello" :> E) :: String
```

```
Overlapping instances for Get String (Tuple '[[Char]])
  arising from a use of 'get'
Matching instances:
  instance Get a (Tuple (a : as))
  instance Get a (Tuple as) => Get a (Tuple (b : as))
In the expression: get ("hello" :> E) :: String
In an equation for 'it': it = get ("hello" :> E) :: String
```

Here, we wanted *get* to dispatch to the first instance, because the first element of the tuple is a *String*. It failed because GHC only inspects the shape of your type – not any equality constraints – when resolving an instance, so it can't tell the two instances apart.

It can be done, but there's a knack to it. The `TypeFamilies` extension enables **closed type families** – type-level functions from types to types – which do understand type equalities. We can write a type-level function which determines whether an element is at the head of a list.

```
data Look = Here | There
type family Where x xs where
  Where x (x ': xs) = Here
  Where x (y ': xs) = There
```

Where can now be put to work in choosing an instance of *Get* for a given tuple. The idea is to delegate to a second class *GetT*, which has an extra parameter for the result of *Where*. GHC can use the value of this extra parameter (which will be either *Here* or *There*) to distinguish the two instances of *GetT*. It's a bit of a hack, but at least it's a well-understood and safe hack.

```
data Proxy a = Proxy

class GetT look r a where    -- T for Tuple
  getT :: proxy look → a → r
instance GetT Here a (Tuple (a ': as))
  where getT _ (x :> xs) = x
instance Get a (Tuple as) ⇒ GetT There a (Tuple (b ': as))
  where getT _ (x :> xs) = get xs

instance (GetT (Where a as) a (Tuple as)) ⇒ Get a (Tuple as) where
  get = getT (Proxy :: Proxy (Where a as))
```

Note the use of *Proxy* – a type whose parameter is irrelevant at runtime – to tell *getT* whether to dispatch to the *Here* or *There* instance. [4]

It works:

```
ghci> get ("hello" :> E) :: String
  "hello"
ghci> get ('a' :> "hello" :> E) :: String
  "hello"
```

Now, at long last, we can do away with boilerplate records and boilerplate instances of *Get* for every predicate. Using a predicate is delightfully simple.

```
test3 = eval' userIsRegistered' (U "lambda" True :> E)  -- True
test4 = eval' userCanEditPost' (u :> p :> E)  -- False
  where
    u = U "forall" False
    p = P {
      isPublic = True,
      canCommentAnonymously = False,
      author = U "lambda" True
    }
```

You get a type error when you forget to provide all the data a predicate needs, albeit with a rather unhelpful error message.

```
test5 = eval' userIsAuthor' (U "pi" False :> E)
```

```
No instance for (GetT (Where Post '[]) Post (Tuple '[]))
  arising from a use of eval'
In the expression: eval' userIsAuthor' (U "pi" False :> E)
In an equation for 'test5':
    test5 = eval' userIsAuthor' (U "pi" False :> E)
```

To me, using *Tuple* to satisfy *Get* constraints feels a bit like using anonymous subclasses in an object-oriented language. In Scala, you might use an anonymous implementation of multiple traits for the same effect:

```
def runPermissionCheck(u : User, p : Post) =
    eval(userCanEditPost, new HasUser with HasPost {
        val user = u
        val post = p
    })
```

Of course, anonymous subclasses rely on subtyping to work. Haskell doesn't feature subtyping, but as you can see it's possible to flex the type system to accommodate programs that make central use of subtyping in other languages.

Code review

We've travelled to the far reaches of the galaxy in our quest to provide a flexible, safe and simple API for predicates.

We began by exploring what you can do with a tree of predicates, folding it up using a higher-order function. Then I showed you how to use type classes in the atomic operations at the leaves of the tree to make predicates of different types composable, letting GHC propagate type class constraints to the root of the tree. This had the effect of deferring the choice of the type a to the use-site of the *Pred*.

Next, we saw how to enforce usage of our new type class-based API, making it impossible to write predicates that refer to a specific type rather than a class. This version made essential use of GADTs, `ConstraintKinds`, and `RankNTypes`. I showed you the general trick to fold up a GADT using polymorphic folding functions and a parameterised return type; then we discovered a technique which uses type classes to simplify the API of such a fold.

Finally, we noted that deferring the choice of a to the use-site of a predicate was inflicting boilerplate code on users. This prompted us to develop an extensible heterogeneous tuple type making use of the `DataKinds` extension, using GADTs to glue together the tuple's type and value. Along the way I showed you a trick using type families to resolve certain kinds of overlapping-instance errors.

In other words, we've used almost all of GHC's modern type system features in this module. I spared you the list of `LANGUAGE` directives at the top of the file, but here it is for completeness:

```
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}
```

I encourage you to read up on the documentation for these extensions!

Would I write code like this in production? Well, maybe. The code is undeniably quite involved and could prove difficult to maintain in the long run. On the other hand, I feel that Haskell has been deliberately designed to support this high level of precision and flexibility. GHC's type system has allowed us to write a program which other languages would reject, and we didn't have to resort to casts, code generation, or reflection. I counted precisely one ugly hack in this article – the instance of *Get* for *Tuple* – and a good proportion of what we wrote was reusable outside of *Pred* and could go in a library.

For a more introductory presentation of predicates (in C♯), you can read my series of blog posts on the topic [2]. More on folding GADTs can be found in a blog post by Williams [3]. My presentation of *Tuple* is based on Kiselyov et al's paper on heterogeneous lists [5]. If you're excited to learn more about programming with types like this, I can recommend as a starting point Lindley and McBride's "Hasochism" paper [6], the introduction to the Agda programming language [7], or the Idris tutorial [8].

References

[1] **Huddle**, `http://www.huddle.com/`

[2] Hodgson, **All About Security**,
`http://tldr.huddle.com/blog/All-about-security/`

[3] Williams, **Fixing GADTs**,
`http://www.timphilipwilliams.com/posts/2013-01-16-fixing-gadts.html`

[4] Yorgey, **Giving Haskell a Promotion**,
`http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/coercible.pdf`

[5] Kiselyov, Lämmel and Schupke, **Strongly Typed Heterogeneous Collections**,
`http://okmij.org/ftp/Haskell/HList-ext.pdf`

[6] Lindley and McBride, **Hasochism**,
`https://personal.cis.strath.ac.uk/conor.mcbride/pub/hasochism.pdf`

[7] Norell and Chapman, **Dependently Typed Programming in Agda**,
`http://www.cse.chalmers.se/~ulfn/darcs/AFP08/LectureNotes/AgdaIntro.pdf`

[8] The Idris Community, **Programming in Idris: A Tutorial**,
`http://eb.host.cs.st-andrews.ac.uk/writings/idris-tutorial.pdf`