Guide to the Haskell Type System

Haskell Types

Andres Löh

14–15 May 2018 —

Well-Typed
The Haskell Consultants

Introduction

Quizzes

An example

```haskell
newtype Question = Q Text
newtype Answer   = A Bool  -- yes or no
```

Quizzes

An example

```haskell
newtype Question = Q Text
newtype Answer   = A Bool  -- yes or no

exampleQ :: [Question]
exampleQ = [ Q "Do you like Haskell?"
           , Q "Do you like dynamic types?"
           ]

exampleA :: [Answer]
exampleA = [ A True
           , A False
           ]
```

Quizzes

An example

```haskell
newtype Question = Q Text
newtype Answer   = A Bool  -- yes or no

exampleQ :: [Question]
exampleQ = [ Q "Do you like Haskell?"
           , Q "Do you like dynamic types?"
           ]

exampleA :: [Answer]
exampleA = [ A True
           , A False
           ]
```

Comments on the design?

Quizzes

contd.

Questions and answers are supposed to be **compatible**,
i.e., of the **same length**.

Quizzes

contd.

Questions and answers are supposed to be **compatible**, i.e., of the **same length**.

Problem gets more pronounced as we continue:

```
type Score   = Int
type Scoring = Answer -> Score

yesno :: Score -> Score -> Scoring
yesno yes no (A b) = if b then yes else no

exampleS :: [Scoring]
exampleS = [ yesno 5 0
           , yesno 0 2
           ]

score :: [Scoring] -> [Answer] -> Score
score ss as = sum (zipWith ($) ss as)
```

Well-Typed

GADTs

From lists to vectors

The types

```
[Question]
[Answer]
[Scoring]
```

provide no information on the length of the list.

From lists to vectors

The types

```
[Question]
[Answer]
[Scoring]
```

provide no information on the length of the list.

What if we had types `Vec n a` of "**vectors**" with exactly `n` elements of type `a` ?

From lists to vectors

The types

```
[Question]
[Answer]
[Scoring]
```

provide no information on the length of the list.

What if we had types `Vec n a` of "**vectors**" with exactly `n` elements of type `a` ?

Numbers at the type level?

Wishful thinking

What we'd like . . .

```
[]  :: Vec 0 a
(:) :: a -> Vec n a -> Vec (1 + n) a
```

Wishful thinking

What we'd like . . .

```
[]  :: Vec 0 a
(:) :: a -> Vec n a -> Vec (1 + n) a
```

A first attempt (in plain Haskell):

```
data Zero                        -- uninhabited type
data Suc n                       -- uninhabited type

newtype Vec n a = Vec [a]        -- phantom type

nil  :: Vec Zero a
cons :: a -> Vec n a -> Vec (Suc n) a

nil              = Vec []
x `cons` Vec xs = Vec (x : xs)
```

Wishful thinking

What we'd like . . .

```
[]  :: Vec 0 a
(:) :: a -> Vec n a -> Vec (1 + n) a
```

A first attempt (in plain Haskell):

```
data Zero                        -- uninhabited type
data Suc n                       -- uninhabited type
newtype Vec n a = Vec [a]        -- phantom type
nil  :: Vec Zero a
cons :: a -> Vec n a -> Vec (Suc n) a
nil             = Vec []
x `cons` Vec xs = Vec (x : xs)
```

Comments on the design?

Phantom types

Short evaluation

Phantom types:

- Useful if you want to expose extra type info in abstract interfaces.
- Examples: FFI (pointers), bindings to C libraries (GUI toolkits).
- Also useful for proxies and tagging (later today).

Not so great here, because we'd like pattern matching.

GADT

```
newtype Vec n a = Vec [a]  -- phantom type

nil  :: Vec Zero a
cons :: a -> Vec n a -> Vec (Suc n) a
```

GADT

```
data Vec :: * -> * -> * where                 -- kind annotation
  Nil  :: Vec Zero a                          -- types of constrs
  Cons :: a -> Vec n a -> Vec (Suc n) a       -- ...
```

GADT

```
data Vec :: * -> * -> * where                    -- kind annotation
  Nil  :: Vec Zero a                             -- types of constrs
  (:*) :: a -> Vec n a -> Vec (Suc n) a          -- ...
infixr 5 :*
```

GADT

```
data Vec :: * -> * -> * where                -- kind annotation
  Nil  :: Vec Zero a                         -- types of constrs
  (:*) :: a -> Vec n a -> Vec (Suc n) a      -- ...
infixr 5 :*
```

Kinds are the types of types.

The kind of normal, unparameterized types is `*` .

GADT

```
data Vec :: * -> * -> * where          -- kind annotation
  Nil  :: Vec Zero a                    -- types of constrs
  (:*) :: a -> Vec n a -> Vec (Suc n) a -- ...
infixr 5 :*
```

Kinds are the types of types.
The kind of normal, unparameterized types is `*` .

GADT syntax lists the types of constructors.
Each constructor must target the defined type (here: `Vec`).
But constructors can **restrict** the parameters.

GADT syntax for “normal” ADTs

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

Constructing vectors

```
> :t 'a' :* 'b' :* Nil
'a' :* 'b' :* Nil :: Vec (Suc (Suc Zero)) Char
```

Natural numbers revisited

We defined:

```
data Zero
data Suc n
```

This simulates natural numbers on the type level:
`Zero` and `Suc` are **types**.

Natural numbers revisited

We defined:

```
data Zero
data Suc n
```

This simulates natural numbers on the type level:
`Zero` and `Suc` are **types**.

We'd normally define natural numbers like this:

```
data Nat = Zero | Suc Nat
```

Here, `Nat` is a **type**, and
`Zero` and `Suc` are **terms**.

Promoting datatypes

Promotion (aka `DataKinds`) allows us to automatically lift (non-GADT) datatypes to the kind level.

We define:

```
data Nat = Zero | Suc Nat
```

We can use `Nat` as a type and `Nat` as a kind.

We can use `Zero` and `Suc` as **terms**, and `'Zero` and `'Suc` as **types**.

The leading quote to indicate promotion is only required to resolve ambiguities and can otherwise be omitted.

Promoting datatypes

contd.

```
data Nat = Zero | Suc Nat
```

Normal interpretation:

```
Nat    :: *
Zero   :: Nat
Suc    :: Nat -> Nat
```

Promoted interpretation:

```
Nat    :: □   -- “is a kind”; syntax not available in GHC
'Zero  :: Nat
'Suc   :: Nat -> Nat
```

Vectors with promoted natural numbers

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec 'Zero a
  (:*) :: a -> Vec n a -> Vec ('Suc n) a
```

Vectors with promoted natural numbers

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
```

Vectors with promoted natural numbers

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
```

Not just more readable,
also rules out types like `Vec Char (Suc Zero)` .

Deriving class instances on vectors

Standard Haskell deriving generally does not work for GADT.

But `StandaloneDeriving` often does!

Deriving class instances on vectors

Standard Haskell deriving generally does not work for GADT.
But `StandaloneDeriving` often does!

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
```

```
deriving instance Show a => Show (Vec n a)
```

Deriving class instances on vectors

Standard Haskell deriving generally does not work for GADT.
But `StandaloneDeriving` often does!

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
```

```
deriving instance Show a => Show (Vec n a)
```

Note:

- For standalone deriving, we have to manually provide the instance context (which makes the job a bit easier for GHC).
- Here, we need `Show a`, but not `Show n` (and with promotion, `Show n` isn't even **kind-correct**).

Back to quizzes

Recap

```haskell
newtype Question = Q Text
newtype Answer   = A Bool  -- yes or no

exampleQ :: [Question]
exampleQ = [ Q "Do you like Haskell?"
           , Q "Do you like dynamic types?"
           ]

exampleA :: [Answer]
exampleA = [ A True
           , A False
           ]
```

Back to quizzes

Now with vectors

```
newtype Question = Q Text
newtype Answer   = A Bool  -- yes or no

exampleQ :: Vec Two Question
exampleQ =    Q "Do you like Haskell?"
           :* Q "Do you like dynamic types?"
           :* Nil

exampleA :: Vec Two Answer
exampleA =    A True
           :* A False
           :* Nil

type Two = Suc (Suc Zero)
```

"Compatibility" of questions and answers is now expressed in the types.

Scoring a quiz

Old version

```haskell
type Score   = Int
type Scoring = Answer -> Score

yesno :: Score -> Score -> Scoring
yesno yes no (A b) = if b then yes else no

exampleS :: [Scoring]
exampleS = [ yesno 5 0
           , yesno 0 2
           ]

score :: [Scoring] -> [Answer] -> Score
score ss as = sum (zipWith ($) ss as)
```

Well-Typed

Scoring a quiz

Now with vectors

```
type Score   = Int
type Scoring = Answer -> Score

yesno :: Score -> Score -> Scoring
yesno yes no (A b) = if b then yes else no

exampleS :: Vec Two Scoring
exampleS =    yesno 5 0
           :* yesno 0 2
           :* Nil

score :: Vec n Scoring -> Vec n Answer -> Score
score ss as = L.sum (V.toList (V.zipWith ($) ss as))
```

Note that `score` requires length-compatible vectors!

We still have to define `toList` and `zipWith` …

Vectors to lists

No surprises here:

```
toList :: Vec n a -> [a]
toList Nil       = []
toList (x :* xs) = x : toList xs
```

Zipping vectors

```
zipWith ::
  (a -> b -> c) -> Vec n a -> Vec n b -> Vec n c
```

All three vectors have the same length!

Zipping vectors

```
zipWith ::
  (a -> b -> c) -> Vec n a -> Vec n b -> Vec n c
```

All three vectors have the same length!

```
zipWith op Nil       Nil       =
  Nil
zipWith op (x :* xs) (y :* ys) =
  (x ‘op‘ y) :* zipWith op xs ys
```

No other cases are required, or even type-correct!

Vectors are functors

If `zipWith` works, `fmap` should be easy:

```
instance Functor (Vec n) where
  fmap :: (a -> b) -> Vec n a -> Vec n b  -- InstanceSigs
  fmap f Nil        = Nil
  fmap f (x :* xs) = f x :* fmap f xs
```

In fact,

```
deriving instance Functor (Vec n)
```

just works.

More examples

Many types of questions

We have:

```
newtype Question = Q Text
```

Well-Typed

Many types of questions

We have:

```
newtype Question = Q Text
```

We want:

```
data Question = Q Text QType
data QType    = QYesNo | QQuant
```

Many types of answers . . .

```haskell
data Question = Q Text QType
data QType    = QYesNo | QQuant
```

Now we need several answers as well:

```haskell
data Answer = AYesNo Bool
            | AQuant Int
```

New compatibility problems

```
exampleQ :: Vec Two Question
exampleQ =    Q "How many type errors?" QQuant
           :* Q "Do you like Haskell?"  QYesNo
           :* Nil
```

```
exampleA :: Vec Two Answer
exampleA =    AYesNo True
           :* AQuant 42
           :* Nil
```

Both vectors have the same length, but they're still not "compatible".

Leads to needless and repeated run-time checking.

GADTs to the rescue

Idea

Let's index questions and answers over their type.

GADTs to the rescue

Idea

Let's index questions and answers over their type.

```
data Question = Q Text QType
data QType    = QYesNo | QQuant


data Answer   = AYesNo Bool
              | AQuant Int
```

GADTs to the rescue

Idea

Let's index questions and answers over their type.

```
data Question a = Q Text (QType a)

data QType :: * -> * where
  QYesNo :: QType Bool
  QQuant :: QType Int

data Answer   = AYesNo Bool
              | AQuant Int
```

GADTs to the rescue

Idea

Let's index questions and answers over their type.

```
data Question a = Q Text (QType a)
data QType :: * -> * where
  QYesNo :: QType Bool
  QQuant :: QType Int
data Answer :: * -> * where
  AYesNo :: Bool -> Answer Bool
  AQuant :: Int  -> Answer Int
```

Singleton types

```
data QType :: * -> * where
  QYesNo :: QType Bool
  QQuant :: QType Int
```

Singleton types

```
data QType :: * -> * where
  QYesNo :: QType Bool
  QQuant :: QType Int
```

The types `QType a` are **singleton types**:

- For each `a`, there's at most one non-bottom value of type `QType a`.
- Singleton types provide a term-level representative for types.
- Singleton types are quite a useful concept in type-level programming that we'll encounter frequently.

New problems

```haskell
data Question a = Q Text (QType a)

data QType :: * -> * where
  QYesNo :: QType Bool
  QQuant :: QType Int

data Answer :: * -> * where
  AYesNo :: Bool -> Answer Bool
  AQuant :: Int  -> Answer Int
```

```haskell
q :: Question Int
q = Q "How many type errors?" QQuant

a :: Answer Int
a = AQuant 0
```

Clearly compatible, but how to build lists or vectors?

Environments (and heterogeneous lists)

Environments

What we need:

- to put things of different types into a list-like structure,
- to keep track of the number of elements and their types in the type system.

Environments

What we need:

- to put things of different types into a list-like structure,
- to keep track of the number of elements and their types in the type system.

A **vector** is indexed by its **length**,
but an **environment** is indexed by **a list of types corresponding to its elements**.

Promoted lists

Fortunately, Haskell allows us to promote the built-in list type.

Normal interpretation:

```
[]  :: * -> *
[]  :: [a]
(:) :: a -> [a] -> [a]
```

Promoted lists

Fortunately, Haskell allows us to promote the built-in list type.

Normal interpretation:

```
[]  :: * -> *
[]  :: [a]
(:) :: a -> [a] -> [a]
```

Promoted interpretation:

```
[]    :: □ -> □
'[]   :: [*]
'(:) ::  *-> [*] -> [*]
```

Here, the quotes are often needed for resolving syntactic ambiguity.

A heterogeneous list

```
data HList :: [*] -> * where
  HNil  :: HList '[]
  HCons :: t -> HList ts -> HList (t ': ts)
infixr 2 `HCons`
```

Defined like this in the `HList` package.

A heterogeneous list

```
data HList :: [*] -> * where
  HNil  :: HList '[]
  HCons :: t -> HList ts -> HList (t ': ts)
infixr 2 `HCons`
```

Defined like this in the `HList` package.

Allows heterogeneous lists, but gives us **too much** flexibility:

```
         Q "How many type errors?" QQuant
 `HCons` AQuant 0
 `HCons` HNil
 :: HList '[Question Int, Answer Int]
```

We want all elements to be questions, or all to be answers . . .

Environments

```
data HList :: [*] -> * where
  HNil  :: HList '[]
  HCons :: t -> HList ts -> HList (t ': ts)
```

Well-Typed

Environments

```
data HList :: [*] -> * where
  HNil  :: HList '[]
  HCons :: t -> HList ts -> HList (t ': ts)
```

```
data Questions :: [*] -> * where
  QNil  :: Questions '[]
  QCons ::
    Question t -> Questions ts -> Questions (t ': ts)
```

Environments

```
data HList :: [*] -> * where
  HNil  :: HList '[]
  HCons :: t -> HList ts -> HList (t ': ts)
```

```
data Questions :: [*] -> * where
  QNil  :: Questions '[]
  QCons ::
    Question t -> Questions ts -> Questions (t ': ts)
```

```
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

Well-Typed

Questions and Answers

```
exampleQ :: Env '[Int, Bool] Question
exampleQ =    Q "How many type errors?" QQuant
           :* Q "Do you like Haskell?"  QYesNo
           :* Nil
```

```
exampleA :: Env '[Bool, Int] Answer
exampleA =    AYesNo True
           :* AQuant 42
           :* Nil
```

It’s now clear from the types that these aren’t compatible.

Deriving instances for environments

This fails:

```
deriving instance Show (Env xs f)
```

And that's to be expected:

- in order to show an environment, we must know `Show (f x)` for all `x` that are elements of `xs` ;
- but how do we express this?

Deriving instances for environments

This fails:

```
deriving instance Show (Env xs f)
```

And that's to be expected:

- in order to show an environment, we must know `Show (f x)` for all `x` that are elements of `xs`;
- but how do we express this?

For now, we can exploit that `Question a` and `Answer a` can be shown without knowing anything about `a`:

```
deriving instance Show (QType a)
deriving instance Show (Question a)
deriving instance Show (Answer a)
deriving instance Show (Env xs Question)
deriving instance Show (Env xs Answer)
```

Scoring with environments

```haskell
type Scoring a = Answer a -> Score
```

does not allow us to form `Env xs Scoring`.

Scoring with environments

```haskell
type Scoring a = Answer a -> Score
```

does not allow us to form `Env xs Scoring` .

```haskell
newtype Scoring a = S (Answer a -> Score)
```

Well-Typed

Scoring with environments

```
type Scoring a = Answer a -> Score
```

does not allow us to form `Env xs Scoring`.

```
newtype Scoring a = S (Answer a -> Score)
```

```
yesno :: Score -> Score -> Scoring Bool
yesno st sf = S (\(AYesNo b) -> if b then st else sf)
quantity :: (Int -> Int) -> Scoring Int
quantity f = S (\(AQuant n) -> f n)
```

Scoring with environment

contd.

```
exampleS :: Env '[Int, Bool] Scoring
exampleS =    quantity negate
           :* yesno    5 0
           :* Nil
```

Well-Typed

Scoring with environment

contd.

```
exampleS :: Env '[Int, Bool] Scoring
exampleS =    quantity negate
           :* yesno    5 0
           :* Nil
```

Direct definition of `score` :

```
score :: Env xs Scoring -> Env xs Answer -> Score
score Nil          Nil       = 0
score (S s :* ss) (a :* as) = s a + score ss as
```

Scoring with environment

contd.

```
exampleS :: Env '[Int, Bool] Scoring
exampleS =     quantity negate
            :* yesno    5 0
            :* Nil
```

Direct definition of `score` :

```
score :: Env xs Scoring -> Env xs Answer -> Score
score Nil          Nil        = 0
score (S s :* ss) (a :* as) = s a + score ss as
```

We had:

```
score ss as = L.sum (V.toList (V.zipWith ($) ss as))
```

Can we recover that?

From environments to lists

We cannot expect to turn arbitrary (heterogeneous) environments into (homogeneous) lists.

```
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

From environments to lists

We cannot expect to turn arbitrary (heterogeneous) environments into (homogeneous) lists.

```
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

But what if `f` is `K a` with:

```
data K a b = K {unK :: a}
```

An `Env xs (K a)` is actually homogeneous:

From environments to lists

We cannot expect to turn arbitrary (heterogeneous) environments into (homogeneous) lists.

```
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

But what if `f` is `K a` with:

```
data K a b = K {unK :: a}
```

An `Env xs (K a)` is actually homogeneous:

```
toList :: Env xs (K a) -> [a]
toList Nil            = []
toList (K x :* xs) = x : toList xs
```

`Env` vs. `HList`

Instantiating `f` to the identity type constructor `I` gives us back heterogeneous lists:

```
data I a = I {unI :: a}
```

```
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

```
data HList :: [*] -> * where
  HNil  :: HList '[]
  HCons :: t -> HList ts -> HList (t ': ts)
```

```
Env xs I ≅ HList xs
```

Zipping environments

For vectors:

```
zipWith :: (a -> b -> c) -> Vec n a  -> Vec n b  -> Vec n c
```

For environments:

```
zipWith :: ...              -> Env xs f -> Env xs g -> Env xs h
```

Let’s try to implement this (in the usual way):

```
zipWith op Nil Nil             = Nil
zipWith op (x :* xs) (y :* ys) =
  (x ‘op‘ y) :* zipWith op xs ys
```

Unfortunately, type inference does not work . . .

Zipping environments

contd.

```
zipWith :: ... -> Env as f -> Env as g -> Env as h
zipWith op Nil Nil                = Nil
zipWith op (x :* xs) (y :* ys) =
  (x ‘op‘ y) :* zipWith op xs ys
```

Well-Typed

Zipping environments

contd.

```
zipWith :: ... -> Env as f -> Env as g -> Env as h
zipWith op Nil Nil                 = Nil
zipWith op (x :* xs) (y :* ys) =
  (x `op` y) :* zipWith op xs ys
```

The function `op` is applied to

```
x :: f a  -- for some 'a' that happens to be in 'as'
y :: g a  -- for the same 'a'
```

Zipping environments

contd.

```
zipWith :: ... -> Env as f -> Env as g -> Env as h
zipWith op Nil Nil              = Nil
zipWith op (x :* xs) (y :* ys) =
  (x ‘op‘ y) :* zipWith op xs ys
```

The function `op` is applied to

```
x :: f a   -- for some ‘a‘ that happens to be in ‘as‘
y :: g a   -- for the same ‘a‘
```

While traversing the lists, `op` is called several times:

- the `f` and `g` are always the same,
- buth `a` changes.

So `op` should be polymorphic in `a` !

Zipping environments

contd.

```
zipWith :: (f a -> g a -> h a)  -- doesn't work
        -> Env as f -> Env as g -> Env as h
```

This is no good.

In a normal (rank-1) polymorphic type:

- the caller can choose all the quantified types,
- the callee must not assume anything about them.

Zipping environments

contd.

```
zipWith :: (forall a. f a -> g a -> h a)
        -> Env as f -> Env as g -> Env as h
```

This is the correct type.

We need a rank-2 polymorphic type:

- the argument itself is polymorphic,
- the caller can't choose, but must provide a polymorphic function,
- the callee can use the argument at different types.

The complete definition

```
zipWith :: (forall a. f a -> g a -> h a)
        -> Env as f -> Env as g -> Env as h
zipWith op Nil Nil               = Nil
zipWith op (x :* xs) (y :* ys) =
  (x ‘op‘ y) :* zipWith op xs ys
```

The scoring function revisited

Direct definition:

```
score :: Env xs Scoring -> Env xs Answer -> Score
score Nil          Nil       = 0
score (S s :* ss) (a :* as) = s a + score ss as
```

Old definition for vectors:

```
score ss as = L.sum (V.toList (V.zipWith ($) ss as))
```

The scoring function revisited

Direct definition:

```
score :: Env xs Scoring -> Env xs Answer -> Score
score Nil          Nil       = 0
score (S s :* ss) (a :* as) = s a + score ss as
```

Old definition for vectors:

```
score ss as = L.sum (V.toList (V.zipWith ($) ss as))
```

New definition with environments:

```
score ss as = L.sum (E.toList (E.zipWith combine ss as))
  where
    combine :: Scoring a -> Answer a -> K Score a
    combine (S f) a = K (f a)
```

Pointing into structures

The situation

- We have an environment of questions and a compatible environment of answers.
- We want to check if there’s any question containing a certain word.
- If so, we want to obtain the corresponding answer and show it.

The situation

- We have an environment of questions and a compatible environment of answers.
- We want to check if there's any question containing a certain word.
- If so, we want to obtain the corresponding answer and show it.

```
task :: Env as Question -> Env as Answer
     -> (Text -> Bool)   -- instead of "containing a certain word"
     -> Maybe String     -- there might be no such question
```

How would we do it normally?

```
task :: [Question] -> [Answer]
     -> (Text -> Bool)
     -> Maybe String
task qs as p = do
  i <- findIndex (\(Q txt _) -> p txt) qs
  let a = as !! i  -- potential crash
  return (show a)
```

How would we do it normally?

```haskell
task :: [Question] -> [Answer]
     -> (Text -> Bool)
     -> Maybe String
task qs as p = do
  i <- findIndex (\(Q txt _) -> p txt) qs
  let a = as !! i  -- potential crash
  return (show a)
```

Can we solve this similarly?

- We need a function like `findIndex` , but what should it return? An `Int` is not suitable.
- We need a function like `(!!)` , ideally one that cannot crash. But depending on index, we get results of different types!

Pointers into environments

We are going to define a new datatype

```
Ptr :: [*] -> * -> *
```

such that `Ptr xs x` represents a "safe" pointer to an element of type `x` in an environment with signature "xs".

Pointers into environments

We are going to define a new datatype

```
Ptr :: [*] -> * -> *
```

such that `Ptr xs x` represents a “safe” pointer to an element of type `x` in an environment with signature “xs”.

Observations and ideas:

- If the signature is empty, there should be **no** valid pointers.
- Otherwise, let’s follow the inductive structure of lists: a pointer can either point at the head of an environment, or at the tail (which requires a pointer into the tail).

Pointers

```
data Ptr :: [*] -> * -> * where
  Head  :: Ptr (x ': xs) x
  Tail  :: Ptr xs y -> Ptr (x ': xs) y
```

Pointers

```
data Ptr :: [*] -> * -> * where
  PZero :: Ptr (x ': xs) x
  PSuc  :: Ptr xs y -> Ptr (x ': xs) y
```

Pointers

```haskell
data Ptr :: [*] -> * -> * where
  PZero :: Ptr (x ': xs) x
  PSuc  :: Ptr xs y -> Ptr (x ': xs) y
```

```haskell
pTwo :: Ptr (x ': y ': z ': zs) z
pTwo = PSuc (PSuc PZero)
```

We start indexing at `0`.

Index `2` requires an environment of length at least `3`.

Performing a lookup

```
(!!) :: Env as f -> Ptr as a -> f a
(x :* xs) !! PZero  = x
(x :* xs) !! PSuc i = xs !! i
```

No cases for the empty environment needed.

No crashes possible.

Finding a pointer

This is more problematic.

Let's start with `findIndex`:

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int
findIndex p [] = Nothing
findIndex p (x : xs)
   | p x        = Just 0
   | otherwise = (1 +) <$> findIndex p xs
```

Finding a pointer

This is more problematic.

Let's start with `findIndex` :

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int
findIndex p [] = Nothing
findIndex p (x : xs)
  | p x         = Just 0
  | otherwise = (1 +) <$> findIndex p xs
```

Now for environments:

```
findPtr ::
  (forall a. f a -> Bool) -> Env as f -> Maybe (Ptr as ...)
findPtr p Nil  = Nothing
findPtr p (x :* xs)
  | p x         = Just PZero
  | otherwise = PSuc <$> findPtr p xs
```

Well-Typed

Hiding types

We don't know the type of the resulting pointer:

```
findPtr ::
  (forall a. f a -> Bool) -> Env as f -> Maybe (Ptr as ...)
```

Yet we do have to provide a result type.

Hiding types

We don't know the type of the resulting pointer:

```
findPtr ::
  (forall a. f a -> Bool) -> Env as f -> Maybe (Ptr as ...)
```

Yet we do have to provide a result type.

```
data SomePtr :: [*] -> * where
  SomePtr :: Ptr as a -> SomePtr as
```

Hiding types

We don't know the type of the resulting pointer:

```
findPtr ::
  (forall a. f a -> Bool) -> Env as f -> Maybe (Ptr as ...)
```

Yet we do have to provide a result type.

```
data SomePtr :: [*] -> * where
  SomePtr :: Ptr as a -> SomePtr as
```

This is called an **existential** type.

When matching on a `SomePtr as`, we know **there exists** a type `a` such that ..., but we don't know the actual type.

Hiding types

We don't know the type of the resulting pointer:

```
findPtr ::
  (forall a. f a -> Bool) -> Env as f -> Maybe (Ptr as ...)
```

Yet we do have to provide a result type.

```
data SomePtr :: [*] -> * where
  SomePtr :: Ptr as a -> SomePtr as
```

This is called an **existential** type.

When matching on a `SomePtr as`, we know **there exists** a type `a` such that . . ., but we don't know the actual type.

```
findPtr ::
  (forall a. f a -> Bool) -> Env as f -> Maybe (SomePtr as)
```

Completing `findPtr`

List version for comparison:

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int
findIndex p [] = Nothing
findIndex p (x : xs)
   | p x        = Just 0
   | otherwise = (1 +) <$> findIndex p xs
```

Version with environments:

```
findPtr ::
  (forall a. f a -> Bool) -> Env as f -> Maybe (SomePtr as)
findPtr p Nil   =  Nothing
findPtr p (x :* xs)
   | p x         =  Just (SomePtr PZero)
   | otherwise  =  (\(SomePtr i) -> SomePtr (PSuc i))
                <$> findPtr p xs
```

Well-Typed

Completing the task

```
task :: [Question] -> [Answer]
     -> (Text -> Bool)
     -> Maybe String
task qs as p = do
  i <- findIndex (\(Q txt _) -> p txt) qs
  let a = as !! i  -- potential crash
  return (show a)
```

Well-Typed

Completing the task

```
task :: [Question] -> [Answer]
     -> (Text -> Bool)
     -> Maybe String
task qs as p = do
  i <- findIndex (\(Q txt _) -> p txt) qs
  let a = as !! i  -- potential crash
  return (show a)
```

```
task :: Env as Question -> Env as Answer
     -> (Text -> Bool)
     -> Maybe String
task qs as p = do
  SomePtr i <- findPtr (\(Q txt _) -> p txt) qs
  let a = as !! i  -- safe
  return (show a)
```

Well-Typed

Establishing invariants

Dealing with the unknown

The problem

In practice, we might want to read questions and answers from a file, the network, or interactively – how can we possibly benefit from all the type safety?

Dealing with the unknown

The problem

In practice, we might want to read questions and answers from a file, the network, or interactively – how can we possibly benefit from all the type safety?

In such a situation:

- We still have to perform a run-time check.
- But we have to perform it once, going from a weakly typed to a strongly typed value in the process.
- Once the additional invariants have been established, we don't need to check them again.

“Typechecking” a list of answers

An example

Let’s assume we’ve obtained a **weakly typed** list of answers:

```
data WAnswer = WAYesNo Bool | WAQuant Int
```

“Typechecking” a list of answers

An example

Let’s assume we’ve obtained a **weakly typed** list of answers:

```
data WAnswer = WAYesNo Bool | WAQuant Int
```

Testing well-formedness in a “normal” setting:

```
chkAnswers :: [WQuestion] -> [WAnswer] -> Bool
```

“Typechecking” a list of answers

An example

Let’s assume we’ve obtained a **weakly typed** list of answers:

```
data WAnswer = WAYesNo Bool | WAQuant Int
```

Testing well-formedness in a “normal” setting:

```
chkAnswers :: [WQuestion] -> [WAnswer] -> Bool
```

In our setting, this becomes:

```
chkAnswers :: Env as Question -> [WAnswer]
           -> Maybe (Env as Answer)
```

Note: `Bool` is replaced with something much more informative!

Implementing `chkAnswers`

```
chkAnswers :: Env as Question -> [WAnswer]
           -> Maybe (Env as Answer)
chkAnswers Nil      []       = Just Nil
chkAnswers (q :* qs) (a : as) = (:*) <$> chkAnswer  q  a
                                     <*> chkAnswers qs as
chkAnswers _        _        = Nothing
```

Implementing `chkAnswers`

```haskell
chkAnswers :: Env as Question -> [WAnswer]
           -> Maybe (Env as Answer)
chkAnswers Nil      []       = Just Nil
chkAnswers (q :* qs) (a : as) = (:*) <$> chkAnswer  q  a
                                     <*> chkAnswers qs as
chkAnswers _        _        = Nothing
```

```haskell
chkAnswer :: Question a -> WAnswer -> Maybe (Answer a)
chkAnswer (Q _ QYesNo) (WAYesNo b) = Just (AYesNo b)
chkAnswer (Q _ QQuant) (WAQuant n) = Just (AQuant n)
chkAnswer _            _           = Nothing
```

Kind polymorphism

Yet more types of questions

Let's assume we want to add another question type for which the answer is also an `Int` :

```
data QType :: * -> * where
  QYesNo :: QType Bool
  QQuant :: QType Int
  QArith :: QType Int
```

```
data Answer :: * -> * where
  AYesNo :: Bool -> Answer Bool
  AQuant :: Int  -> Answer Int
  AArith :: Int  -> Answer Int
```

While this works, it opens up the possibility for incompatibility: we could line up a `QQuant` with an `AArith` .

Why `*` ?

```
data QType :: * -> * where
  QYesNo :: QType Bool
  QQuant :: QType Int
  QArith :: QType Int

data Answer :: * -> * where
  AYesNo :: Bool -> Answer Bool
  AQuant :: Int  -> Answer Int
  AArith :: Int  -> Answer Int
```

Why `*` ?

```
data QType :: * -> * where
  QYesNo :: QType Bool
  QQuant :: QType Int
  QArith :: QType Int

data Answer :: * -> * where
  AYesNo :: Bool -> Answer Bool
  AQuant :: Int  -> Answer Int
  AArith :: Int  -> Answer Int
```

There's not really a need for the index of `Question` , `QType` and `Answer` to be of kind `*` .

In fact, there are many types `a :: *` for which `QType a` or `Answer a` are uninhabited anyway.

Promotion again

```haskell
data QType = QYesNo | QQuant | QArith

data Answer :: QType -> * where
  AYesNo :: Bool -> Answer QYesNo
  AQuant :: Int  -> Answer QQuant
  AArith :: Int  -> Answer QArith
```

Promotion again

```haskell
data QType = QYesNo | QQuant | QArith

data Answer :: QType -> * where
  AYesNo :: Bool -> Answer QYesNo
  AQuant :: Int  -> Answer QQuant
  AArith :: Int  -> Answer QArith
```

So far, so good – but what about `Question` ?

Adapting `Question`

```
data Question (a :: QType) = Q Text...
```

A phantom type is not enough.

We need a GADT to match on, so that we can determine the type at runtime.

Adapting `Question`

```
data Question (a :: QType) = Q Text...
```

A phantom type is not enough.
We need a GADT to match on, so that we can determine the type at runtime.

Let's introduce a **singleton type** for `QType` again:

```
data SQType :: QType -> * where
  SQYesNo :: SQType QYesNo
  SQQuant :: SQType QQuant
  SQArith :: SQType QArith
```

Adapting `Question`

```haskell
data Question (a :: QType) = Q Text...
```

A phantom type is not enough.
We need a GADT to match on, so that we can determine the type at runtime.

Let's introduce a **singleton type** for `QType` again:

```haskell
data SQType :: QType -> * where
  SQYesNo :: SQType QYesNo
  SQQuant :: SQType QQuant
  SQArith :: SQType QArith
```

```haskell
data Question (a :: QType) = Q Text (SQType a)
```

Environments?

```
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

With

```
Question :: QType -> *
```

the type

```
Env '[QYesNo, QQuant] Question
```

is no longer kind-correct.

Do we need a new `Env` type for every kind?

Kind-polymorphic environments

In fact, `Env` works unchanged at a more general kind:

```
data Env :: [k] -> (k -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

Well-Typed

Kind-polymorphic environments

In fact, `Env` works unchanged at a more general kind:

```
data Env :: [k] -> (k -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

The kind of `(->)` is `* -> * -> *`.

However, elements `t` of the list do not appear directly, but only as an argument to `f`.

Kind-polymorphic environments

In fact, `Env` works unchanged at a more general kind:

```
data Env :: [k] -> (k -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

The kind of `(->)` is `* -> * -> *`.

However, elements `t` of the list do not appear directly,

but only as an argument to `f`.

With the generalized kind, we can keep using environments as before.

More kind polymorphism

Other types we've encountered do in fact have more general kinds:

```
Ptr     :: [k] -> k -> *
SomePtr :: [k] -> *
K       ::  *-> k -> *
```

Implementation of GADTs

System FC

GHC’s Core language is called System FC,
An explicitly typed lambda calculus
with **kinds** and **equality constraints**.

System FC

GHC's Core language is called System FC,
An explicitly typed lambda calculus
with **kinds** and **equality constraints**.

Equality constraints also appear in the surface language:

```
a ~ b
```

is a constraint that requires `a` and `b` to be equal.

System FC

GHC's Core language is called System FC,
An explicitly typed lambda calculus
with **kinds** and **equality constraints**.

Equality constraints also appear in the surface language:

```
a ~ b
```

is a constraint that requires `a` and `b` to be equal.

Class constraints are translated to dictionary arguments in Core (and at run-time),
whereas **equality constraints** appear in Core, but are not present at run-time.

GADTs with equality constraints

```
data Env :: [k] -> (k -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
```

can also be written as

```
data Env :: [k] -> (k -> *) -> * where
  Nil  :: (as ~ '[]       ) => Env as f
  (:*) :: (as ~ (t ': ts)) => f t -> Env ts f -> Env as f
```

or even as

```
data Env as f =
                  (as ~ '[]       ) => Nil
   | forall t ts. (as ~ (t ': ts)) => f t :* Env ts f
```

Pattern matching on GADTs

Pattern matching on a GADT constructor reveals the equality constraint(s), which are used to refine the types involved.

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
```

Pattern matching on GADTs

Pattern matching on a GADT constructor reveals the equality constraint(s), which are used to refine the types involved.

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero  ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a
```

Pattern matching on GADTs

Pattern matching on a GADT constructor reveals the equality constraint(s), which are used to refine the types involved.

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero   ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a
```

```
fmap :: (a -> b) -> Vec n a -> Vec n b
fmap f Nil       = Nil
fmap f (x :* xs) = f x :* fmap f xs
```

Pattern matching on GADTs

Pattern matching on a GADT constructor reveals the equality constraint(s), which are used to refine the types involved.

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero   ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a
```

```
fmap :: (a -> b) -> Vec n a -> Vec n b
fmap f Nil       = Nil
fmap f (x :* xs) = f x :* fmap f xs
```

In the first case, `n ~ Zero` .

Pattern matching on GADTs

Pattern matching on a GADT constructor reveals the equality constraint(s), which are used to refine the types involved.

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero  ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a
```

```
fmap :: (a -> b) -> Vec n a -> Vec n b
fmap f Nil        = Nil
fmap f (x :* xs) = f x :* fmap f xs
```

In the second case, `n ~ Suc n'` :

```
             xs :: Vec n'       a
      fmap f xs :: Vec n'       b
f x :* fmap f xs :: Vec (Suc n') b
```

Well-Typed

Pattern matching on GADTs

Pattern matching on a GADT constructor reveals the equality constraint(s), which are used to refine the types involved.

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero  ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a
```

```
fmap :: (a -> b) -> Vec n a -> Vec n b
fmap f Nil        = Nil
fmap f (x :* xs) = f x :* fmap f xs
```

In the second case, `n ~ Suc n'` :

```
              xs :: Vec n'      a
       fmap f xs :: Vec n'      b
f x :* fmap f xs :: Vec n       b
```

Well-Typed

GADTs and type inference

Consider:

```
data X :: * -> * where
  C ::  Int -> X Int
  D ::  X a

f (C n) = [n]
f D     = []
```

What is the type of `f` ?

GADTs and type inference

Consider:

```
data X :: * -> * where
  C ::  Int -> X Int
  D ::  X a

f (C n) = [n]
f D     = []
```

What is the type of `f` ?

```
f :: X a -> [Int]
f :: X a -> [a]
```

None of the two types is an instance of the other.

GADTs and type inference

Consider:

```
data X :: * -> * where
  C ::  Int -> X Int
  D ::  X a

f (C n) = [n]
f D     = []
```

What is the type of `f` ?

```
f :: X a -> [Int]
f :: X a -> [a]
```

None of the two types is an instance of the other.

Functions matching on GADTs do not necessarily have a **principal type**. GHC requires type signatures for such functions.

Producers and singletons

Replicating vectors (or environments)

We've seen a number of functions on GADTs that consume them by pattern matching, like:

```
fmap       :: (a -> b) -> Vec n a -> ...
zipWith    :: (a -> b -> c) -> Env xs f -> Env xs g -> ...
toList     :: Env xs (K a) -> ...
findPtr    :: (forall a. f a -> Bool) -> Env as f -> ...
(!!)       :: Env as f -> Ptr as a -> ...

score      :: Env xs Scoring -> Env xs Answer -> ...
task       :: Env as Question -> Env as Answer -> ...
chkAnswers :: Env as Question -> [WAnswer] -> ...
```

But can we also do something like

```
replicate :: Int -> a -> [a]
```

on vectors or environments?

Using an existential type

Option 1

```
data SomeVec :: * -> * where    -- similar to SomePtr
  SomeVec :: Vec n a -> SomeVec a
```

Using an existential type

Option 1

```
data SomeVec :: * -> * where    -- similar to SomePtr
  SomeVec :: Vec n a -> SomeVec a
```

```
replicate :: Int -> a -> SomeVec a
replicate 0 x = SomeVec Nil
replicate n x = case replicate (n - 1) x of
  SomeVec xs -> SomeVec (x :* xs)
```

Using an existential type

Option 1

```
data SomeVec :: * -> * where    -- similar to SomePtr
  SomeVec :: Vec n a -> SomeVec a
```

```
replicate :: Int -> a -> SomeVec a
replicate 0 x = SomeVec Nil
replicate n x = case replicate (n - 1) x of
  SomeVec xs -> SomeVec (x :* xs)
```

Or even:

```
fromList :: [a] -> SomeVec a
fromList = ...   -- exercise

replicate :: Int -> a -> SomeVec a
replicate n x = fromList (L.replicate n x)
```

Using another vector as template

Option 2

```
replicate :: Vec n b -> a -> Vec n a
replicate Nil x       = Nil
replicate (_ :* ys) x = x :* replicate ys x
```

Or:

```
replicate ys x = fmap (const x) ys
```

Using another vector as template

Option 2

```
replicate :: Vec n b -> a -> Vec n a
replicate Nil x       = Nil
replicate (_ :* ys) x = x :* replicate ys x
```

Or:

```
replicate ys x = fmap (const x) ys
```

But we don't need the elements of the input vector.
What happens if we strip the elements from the `Vec` type?

Singleton natural numbers

```
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
```

```
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

Singleton natural numbers

```haskell
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
```

```haskell
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

```haskell
length :: Vec n a -> SNat n
length Nil       = SZero
length (_ :* xs) = SSuc (length xs)
```

Using an `SNat`

Option 3

```
replicate :: SNat n -> a -> Vec n a
replicate SZero    x = Nil
replicate (SSuc n) x = x :* replicate n x
```

Singletons with class

For singletons, there's only / at most one value per type.

Can we **use the type system** to produce the value?

Singletons with class

For singletons, there's only / at most one value per type.

Can we **use the type system** to produce the value?

```
class SNatI (n :: Nat) where
  sNat :: SNat n

instance SNatI Zero where
  sNat = SZero

instance SNatI n => SNatI (Suc n) where
  sNat = SSuc sNat
```

Using `SNatI`

Option 4

Option 3:

```
replicate :: SNat n -> a -> Vec n a
replicate SZero    x = Nil
replicate (SSuc n) x = x :* replicate n x
```

Now:

```
replicate′ :: SNatI n => a -> Vec n a
replicate′ = replicate sNat
```

Using SNatI

Option 4

Option 3:

```
replicate :: SNat n -> a -> Vec n a
replicate SZero    x = Nil
replicate (SSuc n) x = x :* replicate n x
```

Now:

```
replicate′ :: SNatI n => a -> Vec n a
replicate′ = replicate sNat
```

Example:

```
> zipWith (+) (replicate′ 1) (1 :* 2 :* 3 :* Nil)
2 :* (3 :* (4 :* Nil))
```

Equality

An example

Consider the following list-based code:

```
sameLength :: [a] -> [b] -> Bool
sameLength xs ys = length xs == length ys
```

How can we properly rewrite this to a function on vectors?

```
sameLength :: Vec m a -> Vec n a -> ...
sameLength xs ys = ...
```

An example

Consider the following list-based code:

```
sameLength :: [a] -> [b] -> Bool
sameLength xs ys = length xs == length ys
```

How can we properly rewrite this to a function on vectors?

```
sameLength :: Vec m a -> Vec n a -> ...
sameLength xs ys = ...
```

Using a `Bool` as a result type is not suitable:

```
if sameLength v1 v2 then zipWith op v1 v2 else...
```

fails, but we'd like it to work.

Equality on its own

Using a GADT, we can define a datatype that captures an equality constraint:

```
data (:~:) :: k -> k -> * where
  Refl :: a :~: a  -- or: (a ~ b) => a :~: b
```

This is available (since GHC 7.8) in `Data.Type.Equality`.

Equality on its own

Using a GADT, we can define a datatype that captures an equality constraint:

```
data (:~:) :: k -> k -> * where
  Refl :: a :~: a  -- or: (a ~ b) => a :~: b
```

This is available (since GHC 7.8) in `Data.Type.Equality` .

Now if we have

```
sameLength :: Vec m a -> Vec n a -> Maybe (m :~: n)
```

we can do

```
case sameLength v1 v2 of
  Just Refl -> zipWith op v1 v2
  Nothing   -> ...
```

Completing the definition of `sameLength`

```
sameLength :: Vec m a -> Vec n a -> Maybe (m :~: n)
sameLength xs ys = length xs ==? length ys
```

Recall that `length` returns an `SNat` .

Completing the definition of `sameLength`

```
sameLength :: Vec m a -> Vec n a -> Maybe (m :~: n)
sameLength xs ys = length xs ==? length ys
```

Recall that `length` returns an `SNat`.

So we need:

```
(==?) :: SNat m -> SNat n -> Maybe (m :~: n)
SZero  ==? SZero  = Just Refl
SSuc m ==? SSuc n = case m ==? n of
                      Nothing   -> Nothing
                      Just Refl -> Just Refl  -- sic!
_      ==? _      = Nothing
```

Well-Typed

Decidable equality

The function `(==?)` is also called **semi-decidable equality**, because we return a **proof of equality** on success.

In `Data.Type.Equality`, there's a class for this:

```
class TestEquality (f :: k -> *) where
  testEquality :: f a -> f b -> Maybe (a :~: b)
```

Decidable equality

The function `(==?)` is also called **semi-decidable equality**, because we return a **proof of equality** on success.

In `Data.Type.Equality`, there's a class for this:

```
class TestEquality (f :: k -> *) where
  testEquality :: f a -> f b -> Maybe (a :~: b)
```

```
instance TestEquality SNat where
  testEquality = (==?)
```

Properties of equality

GHC's `~` is an equivalence relation.
We can make it explicit that `:~:` is as well:

```
sym :: (a :~: b) -> (b :~: a)
sym Refl = Refl

trans :: (a :~: b) -> (b :~: c) -> (a :~: c)
trans Refl Refl = Refl
```

Reflexivity is given by `Refl` itself.

Properties of equality

GHC's `~` is an equivalence relation.
We can make it explicit that `:~:` is as well:

```
sym :: (a :~: b) -> (b :~: a)
sym Refl = Refl

trans :: (a :~: b) -> (b :~: c) -> (a :~: c)
trans Refl Refl = Refl
```

Reflexivity is given by `Refl` itself.

```
castWith :: (a :~: b) -> a -> b
castWith Refl x = x

gcastWith :: (a :~: b) -> (a ~ b => r) -> r
gcastWith Refl x = x
```

Type families

Appending two vectors

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

For vectors?

Well-Typed

Appending two vectors

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

For vectors?

```
(++) :: Vec m a -> Vec n a -> Vec...a
Nil       ++ ys = ys
(x :* xs) ++ ys = x :* (xs ++ ys)
```

How to complete the type?

Natural number addition

```
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc m + n = Suc (m + n)
```

Natural number addition

```
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc m + n = Suc (m + n)
```

In a dependently-typed language:

```
(++) :: Vec a m -> Vec a n -> Vec a (m + n)
```

Unfortunately, we **cannot promote functions**.

Use a GADT

GADTs express **relations** on the type level.

Every function is a relation . . .

Use a GADT

GADTs express **relations** on the type level.

Every function is a relation . . .

```
data Plus :: Nat -> Nat -> Nat ->* where
  PlusZero :: Plus Zero n n
  PlusSuc  :: Plus m n n' -> Plus (Suc m) n (Suc n')
```

```
(++) :: Plus m n p -> Vec m a -> Vec n a -> Vec p a
(++) PlusZero    Nil       ys = ys
(++) (PlusSuc p) (x :* xs) ys = x :* (++) p xs ys
```

Use a GADT

GADTs express **relations** on the type level.

Every function is a relation . . .

```
data Plus :: Nat -> Nat -> Nat ->* where
  PlusZero :: Plus Zero n n
  PlusSuc  :: Plus m n n’ -> Plus (Suc m) n (Suc n’)
```

```
(++) :: Plus m n p -> Vec m a -> Vec n a -> Vec p a
(++) PlusZero    Nil       ys = ys
(++) (PlusSuc p) (x :* xs) ys = x :* (++) p xs ys
```

While interesting (and perhaps even useful), it’s quite inconvenient to have to provide a `Plus` argument by hand.

Type family

```
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc m + n = Suc (m + n)
```

```
type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)
```

Type family

```haskell
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc m + n = Suc (m + n)
```

```haskell
type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)
```

```haskell
(++) :: Vec m a -> Vec n a -> Vec (m + n) a
Nil       ++ ys = ys
(x :* xs) ++ ys = x :* (xs ++ ys)
```

Let’s look at the types

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero  ) => Vec n a
  (:*) :: (n ~ Suc n’) => a -> Vec n’ a -> Vec n a

type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)

(++) :: Vec m a -> Vec n a -> Vec (m + n) a
Nil       ++ ys = ys
(x :* xs) ++ ys = x :* (xs ++ ys)
```

Let's look at the types

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero  ) => Vec n a
  (:*) :: (n ~ Suc n') => a -> Vec n' a -> Vec n a

type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)

(++) :: Vec m a -> Vec n a -> Vec (m + n) a
Nil       ++ ys = ys
(x :* xs) ++ ys = x :* (xs ++ ys)
```

In the first case, `m ~ Zero` :

```
ys :: Vec a n
   ~  Vec a (Zero + n)   -- type family
   ~  Vec a (m + n)      -- m ~ Zero
```

Let’s look at the types

```
data Vec :: Nat -> * -> * where
  Nil  :: (n ~ Zero  ) => Vec n a
  (:*) :: (n ~ Suc n’) => a -> Vec n’ a -> Vec n a

type family (+) (m :: Nat) (n :: Nat) :: Nat where
  Zero  + n = n
  Suc m + n = Suc (m + n)

(++) :: Vec m a -> Vec n a -> Vec (m + n) a
Nil       ++ ys = ys
(x :* xs) ++ ys = x :* (xs ++ ys)
```

In the second case, `m ~ Suc m’` :

```
x :* (xs ++ ys) :: Vec a Suc (m’ + n)
                 ~ Vec a (Suc m’ + n)   -- type family
                 ~ Vec a (m + n)        -- m ~ Suc m’
```

Well-Typed

Proving properties

Defining `reverse` on vectors

```
reverse :: Vec n a -> Vec n a
reverse xs = go xs Nil  -- fails
  where
    go :: Vec m a -> Vec n a -> Vec (m + n) a
    go Nil       acc = acc
    go (x :* xs) acc = go xs (x :* acc)  -- fails
```

Unfortunately, this does not type-check. Why?

Two simple properties

```
thmPlusZero :: SNat n -> (n + Zero) :~: n
thmPlusZero SZero    = Refl
thmPlusZero (SSuc s) = gcastWith (thmPlusZero s) Refl
```

```
thmPlusSuc :: SNat m -> SNat n
           -> (m + Suc n) :~: (Suc (m + n))
thmPlusSuc SZero    _ = Refl
thmPlusSuc (SSuc s) n = gcastWith (thmPlusSuc s n) Refl
```

Using the properties

```
reverse :: Vec n a -> Vec n a
reverse xs =
  gcastWith (thmPlusZero (length xs)) $ go xs Nil
  where
    go :: Vec m a -> Vec n a -> Vec (m + n) a
    go Nil       acc = acc
    go (x :* xs) acc =
      gcastWith (thmPlusSuc (length xs) (length acc)) $
      go xs (x :* acc)
```

More on type families

Associated types

Type families can also be associated with a class:

```
class Sequence (as :: *) where
  type Elt as :: *
  filter :: (Elt as -> Bool) -> as -> as
  ...
```

Associated types

Type families can also be associated with a class:

```
class Sequence (as :: *) where
  type Elt as :: *
  filter :: (Elt as -> Bool) -> as -> as
  ...
```

```
instance Sequence [a] where
  type Elt [a] = a
  filter = L.filter

instance Sequence Text where
  type Elt Text = Char
  filter = T.filter
```

Mainly a syntactic difference.

Type family implementation

Type families introduce new symbols and associated equality constraints to System FC:

```
type family (+) (m :: Nat) (n :: Nat) :: Nat
type instance Zero  + n = n
type instance Suc m + n = Suc (m + n)
```

Type family implementation

Type families introduce new symbols and associated equality constraints to System FC:

```
type family (+) (m :: Nat) (n :: Nat) :: Nat
type instance Zero  + n = n
type instance Suc m + n = Suc (m + n)
```

introduces `(+)` with

```
Zero  + n ~ n
Suc m + n ~ Suc (m + n)
```

Type family implementation

Type families introduce new symbols and associated equality constraints to System FC:

```
type family (+) (m :: Nat) (n :: Nat) :: Nat
type instance Zero  + n = n
type instance Suc m + n = Suc (m + n)
```

introduces `(+)` with

```
Zero  + n ~ n
Suc m + n ~ Suc (m + n)
```

Type families:

- must always be fully applied.
- are open.
- must not have overlapping cases (but since GHC 7.8, closed type families exist).

Injectivity

Different representations of data

An example

```
class Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int

instance Compactable Int
instance Compactable a => Compactable [a]
```

A strange error

Couldn't match type ‘`Compact a`’ with ‘`Compact a0`’
Expected type: ‘`Compact a -> Int`’
Actual type: ‘`Compact a0 -> Int`’
NB: ‘`Compact`’ is a type function, and may not be injective
The type variable ‘`a0`’ is amgiguous
In the ambiguity check for ‘`size`’
To defer the ambiguity check to use sites,
  enable `AllowAmbiguousTypes`

We can try enabling `AllowAmbiguousTypes`, but then …

A strange error

contd.

```
test = size (compact [1, 2, 3])
```

A strange error

contd.

```
test = size (compact [1, 2, 3])
```

Couldn't match expected type '`Compact a0`'
  with actual type '`Compact [t0]`'
NB: '`Compact`' is a type function, and may not be injective
The type variables '`a0`', '`t0`' are ambiguous
In the first argument of '`size`', namely '`(compact [1, 2, 3])`'
In the expression: `size (compact [1, 2, 3])`

A strange error

contd.

```
test = size (compact [1, 2, 3])
```

Couldn't match expected type '`Compact a0`'
  with actual type '`Compact [t0]`'
NB: '`Compact`' is a type function, and may not be injective
The type variables '`a0`', '`t0`' are ambiguous
In the first argument of '`size`', namely '`(compact [1, 2, 3])`'
In the expression: `size (compact [1, 2, 3])`

```
test = size (compact ([1, 2, 3] :: [Int]) :: Compact [Int])
```

is not improving anything.

Explaining the error

```
test = size (compact [1, 2, 3] :: Compact [Int])
```

```
size               :: Compactable a => Compact a -> Int
compact [1, 2, 3] :: Compact [Int]
```

Explaining the error

```
test = size (compact [1, 2, 3] :: Compact [Int])
```

```
size               :: Compactable a => Compact a -> Int
compact [1, 2, 3] :: Compact [Int]
```

So we have to unify:

```
Compact [Int] ~ Compact a
```

It seems like `a ~ [Int]` is an obvious solution, but is it the only one?

Type families need not be injective

```
type Compact [a] = Array Int (Compact a)
type Compact Int = Int
```

```
newtype Count = Count Int
type Compact Count = Int
```

Type families need not be injective

```
type Compact [a] = Array Int (Compact a)
type Compact Int = Int
```

```
newtype Count = Count Int
type Compact Count = Int
```

Now:

```
Compact [Int] ~ Array Int Int ~ Compact [Count]
```

Well-Typed

Injectivity

In general, a function `f` is called **injective** if `f x ~ f y` implies `x ~ y`.

Injectivity

In general, a function `f` is called **injective** if `f x ~ f y` implies `x ~ y`.

Datatypes (both `data` and `newtype`) are **always** injective, but type families (and type synonyms) are generally not.

Recognizing problematic functions

```
class Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

Recognizing problematic functions

```
class Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

In `size`, the type variable `a` appears only as an argument to a type family – it seems impossible to use this function in practice.

Making the type family injective

Solution 1

```
class Compactable (a :: *) where
  type Compact a = (r :: *) | r -> a
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int

instance Compactable Int
instance Compactable a => Compactable [a]

test = size (compact [1, 2, 3] :: Compact [Int])
```

Making the type family injective

Solution 1

```haskell
class Compactable (a :: *) where
  type Compact a = (r :: *) | r -> a
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int

instance Compactable Int
instance Compactable a => Compactable [a]

test = size (compact [1, 2, 3] :: Compact [Int])
```

The function `test` is now accepted. GHC enforces injectivity.

Straight-forward, but not all type families are injective, so not always an option. Requires GHC 8; syntax may be subject to change.

Redesigning the class hierarchy

Solution 2

```haskell
class Size (Compact a) => Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a

class Size a where
  size :: a -> Int
```

Probably the best solution in this situation.

Redesigning the class hierarchy

Solution 2

```
class Size (Compact a) => Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
class Size a where
  size :: a -> Int
```

Probably the best solution in this situation.

Now

```
test = size (compact ([1, 2, 3] :: [Int]))
```

typechecks as long as `Size (Compact [Int])` holds.

Using a proxy

Solution 3

```
data Proxy (a :: k) = Proxy

class Compactable (a ::*) where
  type Compact a ::*
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Proxy a -> Compact a -> Int
```

The additional argument is annoying, but this always works.

```
test = size (Proxy :: Proxy [Int])
  (compact ([1, 2, 3] :: [Int]))
```

Using a proxy

Solution 3

```
data Proxy (a :: k) = Proxy

class Compactable (a ::*) where
  type Compact a ::*
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Proxy a -> Compact a -> Int
```

The additional argument is annoying, but this always works.

```
test = size (Proxy :: Proxy [Int])
  (compact ([1, 2, 3] :: [Int]))
```

```
data Tagged (a :: k) b = Tagged b

size :: Tagged a (Compact a) -> Int  -- another option
```

Well-Typed

Using explicit type application

Solution 4

```
class Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

```
test = size @[Int]
  (compact ([1, 2, 3] :: [Int]))
```

Requires GHC 8.

Well-Typed

Writing an inverse

Solution 4

If the function actually is injective,
we can “prove” it by writing an inverse:

```
type family Uncompact (a :: *) :: *

class (Uncompact (Compact a) ~ a)
        => Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

Extra work to define `Uncompact`.

Writing an inverse

Solution 4

If the function actually is injective,
we can “prove” it by writing an inverse:

```
type family Uncompact (a :: *) :: *

class (Uncompact (Compact a) ~ a)
        => Compactable (a :: *) where
  type Compact a :: *
  compact   :: a -> Compact a
  uncompact :: Compact a -> a
  size      :: Compact a -> Int
```

Extra work to define `Uncompact` .

But now, by applying `Uncompact` , we can actually solve

```
Compact a ~ Compact b
```

Data families

Next to

```
type family X...
```

there's also

```
data family X...
```

that allows

```
data instance...
newtype instance...
```

(And associated datatypes correspondingly.)

Tempting because they're always injective – not useful if you work with types that already exist.

Generalizing singleton types

We can use a **kind-indexed** data family to make singletons less ad-hoc.

Before:

```
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

Generalizing singleton types

We can use a **kind-indexed** data family to make singletons less ad-hoc.

Before:

```
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

Now:

```
data family Sing (a :: k)
data instance Sing (n :: Nat) where
  SZero :: Sing Zero
  SSuc  :: Sing n -> Sing (Suc n)
```

Constraint kinds

Classes have kinds

```
Eq         :: * -> Constraint
Functor    :: (* -> *) -> Constraint
MonadState :: * -> (* -> *) -> Constraint
```

Classes have kinds

```
Eq        :: * -> Constraint
Functor   :: (* -> *) -> Constraint
MonadState :: * -> (* -> *) -> Constraint
```

By viewing constraints as kind, we can e.g.

- define class synonyms using `type`,
- parameterize types and classes over constraints,
- define constraint families.

Restricted monads

The classic example

Sets as defined in `Data.Set` aren't monads:

```
returnSet :: a -> Set a
bindSet :: Ord a => Set a -> (a -> Set b) -> Set b
```

The `Ord` constraint does not fit.

Restricted monads

The classic example

Sets as defined in `Data.Set` aren't monads:

```
returnSet :: a -> Set a
bindSet :: Ord a => Set a -> (a -> Set b) -> Set b
```

The `Ord` constraint does not fit.

```
class RMonad (c :: * -> Constraint) (m :: * -> *) where
  return :: c a => a -> m a
  (>>=)  :: c a => m a -> (a -> m b) -> m b

instance RMonad Ord Set
```

Showing environments

```
type family
  All (c :: k -> Constraint) (xs :: [k]) :: Constraint
  where
  All c '[]        = ()
  All c (x ': xs) = (c x, All c xs)
type family Map (f :: k1 -> k2) (xs :: [k1]) :: [k2]
  where
  Map f '[]        = '[]
  Map f (x ': xs) = (f x) ': (Map f xs)
```

Showing environments

```
type family
  All (c :: k -> Constraint) (xs :: [k]) :: Constraint
  where
  All c '[]       = ()
  All c (x ': xs) = (c x, All c xs)
type family Map (f :: k1 -> k2) (xs :: [k1]) :: [k2]
  where
  Map f '[]       = '[]
  Map f (x ': xs) = (f x) ': (Map f xs)
```

```
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
deriving instance All Show (Map f xs) => Show (Env xs f)
```

More

What we haven't (explicitly) covered

- Functional dependencies
- Type literals
- Higher-order type families
- Indexed / parameterized monads
- Open type families
- Roles
- ...