More Generalised Algebraic Datatypes

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK

2018-02-28

Goals

- Motivate a number of type system extensions.
- In particular, look at Generalised Algebraic Datatypes (GADTs).

Recap: environments

```
data Env :: [*] -> (* -> *) -> * where
  Nil  :: Env '[] f
  (:*) :: f t -> Env ts f -> Env (t ': ts) f
infixr 5 :*
```

Recap: 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
```

Recap: scoring preparations

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

```haskell
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

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

Scoring with environment

```
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
```

Can we recover our old definition?

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

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:

```
newtype K a b = K {unK :: a}  -- like Const, just shorter
```

From environments to lists (contd.)

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

From environments to lists (contd.)

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

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

From environments to lists (contd.)

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

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

In fact, such an environment is isomorphic to a vector of the length of the type-level list.

Env vs. HList

```
newtype I a = I {unI :: a}  -- like Identity
```

```
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` vs. `HList`

```
newtype I a = I {unI :: a}  -- like Identity
```

```
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
```

Exercise

Define these isomorphisms:

```
envToHList :: Env xs I -> HList xs
hListToEnv :: HList xs -> Env xs I
```

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

```
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
```

Zipping environments

```
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

```
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,
- but `a` changes, so `op` should be polymorphic in `a`.

Zipping environments

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

We need a rank-2 polymorphic type again. Recall:

- 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?

```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)
```

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)
```

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

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

```
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

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

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
```

For environments, we have a problem:

```
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
```

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 (we've seen these with free monads as well).

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

Completing `findPtr`

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
```

This is still very close to the version for lists.

Completing the task

```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)
```

Completing the task

```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)
```

```haskell
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)
```

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

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

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

"Typechecking" a list of answers

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

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`

```
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
```

```
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 `*`?

```haskell
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 `*`?

```haskell
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

```
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`

```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.

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`

```
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
```

```
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
```

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 -> *
```

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] -> ...
```

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?

Option 1: Using an existential type

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

Option 1: Using an existential type

```
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)
```

Option 1: Using an existential type

```
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)
```

Option 2: Using another vector as template

```
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
```

Option 2: Using another vector as template

```
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

```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)
```

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)
```

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

Option 3: Using an SNat

```
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
```

Option 4: Using `SNatI`

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
```

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:

```
GHCi> 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

We can capture an equality constraint in a GADT:

```
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

We can capture an equality constraint in a GADT:

```
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    -- sic!
                      Just Refl -> Just Refl  -- sic!
_      ==? _      = 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`.

So we need:

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

(Why does `SSuc m ==? SSuc n = m ==? n` not work?)

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 = (\Refl -> Refl) <$> m ==? n



_      ==? _      = Nothing
```

This is slightly nicer, perhaps.

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
```