Generalised Algebraic Datatypes

Haskell and Cryptocurrencies

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

2018-02-27

Goals

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

Recap

- We have encountered GADTs already, as an intermediate step in the context of the free monads.
- In this lecture, we will see more and new examples and talk about GADTs in more detail.

Motivation

Quizzes

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

Quizzes

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

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

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

Quizzes

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

```haskell
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

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

Quizzes

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)
```

Capturing invariants

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

Phantom types:

- Useful if you want to expose extra type info in abstract interfaces.
- Example: Modelling typed C pointers (internally, just an address, but we want to remember the type).
- Also useful for proxies and tagging – which we will introduce later.

Phantom types

Phantom types:

- Useful if you want to expose extra type info in abstract interfaces.
- Example: Modelling typed C pointers (internally, just an address, but we want to remember the type).
- Also useful for proxies and tagging – which we will introduce later.

Not so great here, because we'd like type-awareness in pattern matching.

```haskell
newtype Vec n a = Vec [a]  -- phantom type
nil  :: Vec Zero a
cons :: a -> Vec n a -> Vec (Suc n) a
```

We provide a kind annotation, and list the types of the constructors.

GADT

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

We provide a kind annotation, and list the types of the constructors.

```
data Vec :: * -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
infixr 5 :*
```

We provide a kind annotation, and list the types of the constructors.

GADT

```
data Vec :: * -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
infixr 5 :*
```

We provide a kind annotation, and list the types of the 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
```

GADT syntax for “normal” ADTs

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

- Every datatype can be written in GADT syntax.
- For normal datatypes, the result type is never restricted.

Constructing vectors

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

Constructing vectors

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

Unfortunately, we don't have a `Show` instance, and using `deriving Show` does not work for GADTs.

Standalone deriving

However, in this case we can recover by using the `StandaloneDeriving` extension:

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

Standalone deriving

However, in this case we can recover by using the `StandaloneDeriving` extension:

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

This is a bit easier for GHC because we have to manually provide the instance context. For example, in this case we need `Show a` but *not* `Show n`.

Standalone deriving

However, in this case we can recover by using the `StandaloneDeriving` extension:

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

This is a bit easier for GHC because we have to manually provide the instance context. For example, in this case we need `Show a` but *not* `Show n`.

```
GHCi> 'a' :* 'b' :* Nil
'a' :* ('b' :* Nil)
```

Derived `Show` instances print unnecessary parentheses, but at least it works.

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

```
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

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

Vectors with promoted natural numbers

```haskell
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

We can still use `StandaloneDeriving` to derive a `Show` instance for vectors.

Deriving class instances on vectors

We can still use `StandaloneDeriving` to derive a `Show` instance for vectors.

```
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

We can still use `StandaloneDeriving` to derive a `Show` instance for vectors.

```
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:

- We still do not need `Show n`, and with promotion, it is no longer even *kind-correct*, because `Show` is parameterized over types of kind `*`.

Using GADTs

Back to quizzes

```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

```
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

```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)
```

Scoring a quiz

```
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!

Functions on vectors

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

No surprises for `toList`:

```
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
  fmap f Nil        = Nil
  fmap f (x :* xs) = f x :* fmap f xs
```

In fact,

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

just works.

A look at the internals

System FC

One of the extensions that GHC's Core language has over System F are equality constraints.

System FC

One of the extensions that GHC's Core language has over System F are equality constraints.

Equality constraints also appear in the surface language (i.e., in Haskell itself):

```
a ~ b
```

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

System FC

One of the extensions that GHC's Core language has over System F are equality constraints.

Equality constraints also appear in the surface language (i.e., in Haskell itself):

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

can also be written as

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

Pattern matching on GADTs

```
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 reveals equality constraints:

Pattern matching on GADTs

```
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 reveals equality constraints:

In the first case, `n ~ Zero`.
Therefore, `Nil` is ok as the result.

Pattern matching on GADTs

```
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 reveals equality constraints:

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

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]
```

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 (contd.)

Functions matching on GADTs do not necessarily have a *principal type*.

GHC requires type signatures for such functions.

Exercises

```haskell
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a
infixr 5 :*
deriving instance Show a => Show (Vec n a)
```

Define for vectors:

```haskell
head
tail
minimum
foldr   -- optional; if you have time
```

More examples

Many types of questions

We have:

```
newtype Question = Q Text
```

Many types of questions

We have:

```haskell
newtype Question = Q Text
```

We want:

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

Many types of answers ...

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

Now we need several answers as well:

```
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

```haskell
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

```haskell
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:

```haskell
          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)
```

Environments

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

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

Environments

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

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

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

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:

```haskell
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

```
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)
```

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
```

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
```