Introduction to Type-Level and Generic Programming in Haskell

CUFP 2015

Andres Löh

3 September 2015

Well-Typed
The Haskell Consultants

Datatype-generic programming

Express algorithms that make use of the structure of datatypes

Datatype-generic programming

Express algorithms that make use of the structure of datatypes

```
eqA :: A -> A -> Bool
```

A class

```
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

where `from` and `to` are inverses.

A class

```
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

where `from` and `to` are inverses.

```
geq :: Generic a => Rep a -> Rep a -> Bool
```

A class

```haskell
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

where `from` and `to` are inverses.

```haskell
geq :: Generic a => Rep a -> Rep a -> Bool
```

```haskell
eq :: Generic a => a -> a -> Bool
eq x y = geq (from x) (from y)
```

Choices

Much flexibility in the details, in particular the definition of `Rep`.

Choices

Much flexibility in the details, in particular the definition of `Rep`.

The choice of `Rep` determines expressive power and flavour of generic programs.

Choices

Much flexibility in the details, in particular the definition of `Rep`.

The choice of `Rep` determines expressive power and flavour of generic programs.

In this tutorial: generics-sop.

Applications

- (De-)serialization
- Data generation
- Data traversals
- Data navigation
- …

The generics-sop view on data, informally

Sample datatypes

```haskell
data Maybe a    = Nothing | Just a
data Either a b = Left a | Right b
data Group      = Group Char Bool Int
data Expr       = NumL Int
                | BoolL Bool
                | Add Expr Expr
                | If Expr Expr Expr
```

Sample datatypes

```
data Maybe a    = Nothing | Just a
data Either a b = Left a | Right b
data Group      = Group Char Bool Int
data Expr       = NumL Int
                | BoolL Bool
                | Add Expr Expr
                | If Expr Expr Expr
```

- Choice between constructors,
- each with a sequence of arguments.

Sample datatypes

```
data Maybe a    = Nothing | Just a
data Either a b = Left a | Right b
data Group      = Group Char Bool Int
data Expr       = NumL Int
                | BoolL Bool
                | Add Expr Expr
                | If Expr Expr Expr
```

- Choice between constructors,
- each with a sequence of arguments.

$$C_i\ x_0 \ldots x_{n_i - 1}$$

The plan

$$C_i\ x_0 \ldots x_{n_i-1}$$

- Choice between constructors modelled as an *n*-ary sum.
- Sequence of fields modelled as an *n*-ary product.

We’ll need Haskell type-level programming concepts along the way.

Extensions, extensions

```
DataKinds
GADTs
TypeOperators
TypeFamilies
RankNTypes
ConstraintKinds
MultiParamTypeClasses
UndecidableInstances
StandaloneDeriving
ScopedTypeVariables
PolyKinds
FlexibleInstances
FlexibleContexts
DefaultSignatures
```

Overall plan

- Learn about *n*-ary products and *n*-ary sums.
- Along the way, discuss everything we need in terms of Haskell type-level programming features.
- Representing datatypes using generics-sop.
- Applications.
- Handling metadata.