

Introduction to Type-Level and Generic Programming in Haskell

CUFP 2016

Andres Löh

22 September 2016



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

```
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
```

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

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

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

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

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

```
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.

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

The plan

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

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

The plan

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

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

$SOP \ (S \dots Z \ (I \ x_0 \ :* \dots \ :* \ I \ x_{n_i-1} \ :* \ Nil))$

Extensions, extensions

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

Overall plan

- ▶ Learn about n -ary products and n -ary sums.
- ▶ Example generic functions.
- ▶ Explain internals, discuss type-level programming features.
- ▶ More generic functions.
- ▶ Handling metadata.