True Sums of Products

Workshop on Generic Programming 2014

Edsko de Vries, Andres Löh

31 August 2014

Overview

A new approach to datatype-generic programming (in Haskell).

Overview

A new approach to datatype-generic programming (in Haskell).

Notable features:

- Faithful representation of datatypes
  as *n*-ary sums of *n*-ary products.
- A library of high-level combinators
  to encourage concise and reusable functions.
- The separation of metadata in the representation.

Overview

A new approach to datatype-generic programming (in Haskell).

Notable features:

- Faithful representation of datatypes
  as *n*-ary sums of *n*-ary products.
- A library of high-level combinators
  to encourage concise and reusable functions.
- The separation of metadata in the representation.

Library available on Hackage: `generics-sop`

Datatype-generic programming

- We want to write functions over a type A.

Datatype-generic programming

- We want to write functions over a type `A`.
- We can convert between `A` and an isomorphic `Rep A`.

Datatype-generic programming

- We want to write functions over a type `A`.
- We can convert between `A` and an isomorphic `Rep A`.
- All `Rep` types share a common structure.

Datatype-generic programming

- We want to write functions over a type `A`.
- We can convert between `A` and an isomorphic `Rep A`.
- All `Rep` types share a common structure.
- We can thus define functions over `Rep` types, and they'll work for `A` as well as any other representable type.

Example

Generating a default value

```
class Def a where def :: a
```

Generating a default value

```
class Def a where def :: a
```

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

Generating a default value

```
class Def a where def :: a
```

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

Generating a default value

```
class Def a where def :: a
```

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
Z (cpure_NP (Proxy :: Proxy Def) (I def))  ≈ Z [def]
```

Generating a default value

```
class Def a where def :: a
```

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
to (Z (cpure_NP (Proxy :: Proxy Def) (I def))) ≈ LInt 0
```

Assuming:

```
instance Def Int where def = 0
```

Generating a default value

```
class Def a where def :: a
```

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
instance Def Expr where def = gdef
```

Sum of products representation

Our generic representation

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

A sum of products:

- the sum is choice between constructors,
- depending on choice, the product is the sequence of arguments.

Our generic representation

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

Example values:

```
LInt 0
If (LBool True) (LInt 3) (LInt 5)
```

Our generic representation

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

Example values:

```
LInt 0
If (LBool True) (LInt 3) (LInt 5)
```

“Generically”:

```
C0 [0]
C2 [LBool True, LInt 3, LInt 5]
```

Our generic representation

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

Example values:

```
LInt 0
If (LBool True) (LInt 3) (LInt 5)
```

“Generically”:

```
C0 [0]
C2 [LBool True, LInt 3, LInt 5]
```

Actually:

```
Z         $ I 0 :* Nil
S . S . Z $ I (LBool True) :* I (LInt 3) :* I (LInt 5) :* Nil
```

Types and their representations

```
class SingI (Code a) ⇒ Generic a where
  type Code a :: [[*]]
  from :: a → Rep a
  to   :: Rep a → a
type Rep = SOP I (Code a)
```

Types and their representations

```
class SingI (Code a) ⇒ Generic a where
  type Code a :: [[∗]]
  from :: a → Rep a
  to   :: Rep a → a
type Rep = SOP I (Code a)
```

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
from (LInt 0) = Z          $ I 0 :∗ Nil
from (If (LBool True) (LInt 3) (LInt 5))
              = S . S . Z $ I (LBool True) :∗ I (LInt 3) :∗ I (LInt 5) :∗ Nil
```

Codes

```
class SingI (Code a) ⇒ Generic a where
  type Code a :: [[∗]]
  ...
```

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
instance Generic Expr where
  type Code Expr =  '[ '[Int]
                     , '[Bool]
                     , '[Expr, Expr, Expr]
                     ]
  ...
```

Products

```
Z           $ I 0 :∗ Nil
S . S . Z $ I (LBool True) :∗ I (LInt 3) :∗ I (LInt 5) :∗ Nil
```

Products

```
Z         $ I 0 :* Nil
S . S . Z $ I (LBool True) :* I (LInt 3) :* I (LInt 5) :* Nil
```

```
Nil                          :: NP f '[]
I 0 :* Nil                   :: NP I '[Int]
I (LBool True) :* I (LInt 3) :* I (LInt 5) :* Nil
                             :: NP I '[Expr, Expr, Expr]
```

```
newtype I a = I a
```

Products

```
Z          $ I 0 :* Nil
S . S . Z $ I (LBool True) :* I (LInt 3) :* I (LInt 5) :* Nil
```

```
Nil                          :: NP f '[]
I 0 :* Nil                   :: NP I '[Int]
I (LBool True) :* I (LInt 3) :* I (LInt 5) :* Nil
                             :: NP I '[Expr, Expr, Expr]
```

```
newtype I a = I a
```

```
data NP :: (k → *) → [k] → * where
  Nil  :: NP f '[]
  (:*) :: f x → NP f xs → NP f (x ': xs)
```

Special cases

```
newtype I a = I a
```

A `NP I xs` is like a heterogeneous list or a nested pair.

```
newtype K a b = K a
```

A `NP (K a) xs` is a homogeneous list of statically known length.

Sums

```
Z           $ I 0 :∗ Nil
S . S . Z $ I (LBool True) :∗ I (LInt 3) :∗ I (LInt 5) :∗ Nil
```

Sums

```
Z         $ I 0 :∗ Nil
S . S . Z $ I (LBool True) :∗ I (LInt 3) :∗ I (LInt 5) :∗ Nil
```

```
Z         :: f x → NS f (x ′: xs)
S . Z     :: f y → NS f (x ′: y ′: xs)
S . S . Z :: f z → NS f (x ′: y ′: z ′: xs)
```

Sums

```
Z         $ I 0 :∗ Nil
S . S . Z $ I (LBool True) :∗ I (LInt 3) :∗ I (LInt 5) :∗ Nil
```

```
Z         :: f x → NS f (x ′: xs)
S . Z     :: f y → NS f (x ′: y ′: xs)
S . S . Z :: f z → NS f (x ′: y ′: z ′: xs)
```

```
data NS :: (k → ∗) → [k] → ∗ where
  Z :: f x → NS f (x ′: xs)
  S :: NS f xs → NS f (x ′: xs)
```

Sums

```
Z         $ I 0 :* Nil
S . S . Z $ I (LBool True) :* I (LInt 3) :* I (LInt 5) :* Nil
```

```
Z         :: f x → NS f (x ': xs)
S . Z     :: f y → NS f (x ': y ': xs)
S . S . Z :: f z → NS f (x ': y ': z ': xs)
```

```
data NS :: (k → *) → [k] → * where
  Z :: f x → NS f (x ': xs)
  S :: NS f xs → NS f (x ': xs)
```

```
Z $ I 0 :* Nil :: NS (NP I) ( '[Int] ': xs)
```

```
type SOP (f :: k → *) (xss :: [[k]]) = NS (NP f) xss
```

Putting everything together

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
type Code Expr =  '[ '[Int]
                   , '[Bool]
                   , '[Expr, Expr, Expr]
                   ]
```

```
type Rep a = SOP I (Code a)
```

```
type SOP f xss = NS (NP f) xss
```

```
Z         $ I 0 :* Nil                                        :: Rep Expr
S . S . Z $ I (LBool True) :* I (LInt 3) :* I (LInt 5) :* Nil :: Rep Expr
```

The complete instance

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
instance Generic Expr where
  type Code Expr = '[ '[Int], '[Bool], '[Expr, Expr, Expr] ]
  from :: Expr → Rep Expr
  from (LInt  n)     = Z     (I n :∗ Nil)
  from (LBool b)     = S (Z  (I b :∗ Nil))
  from (If    c t e) = S (S (Z (I c :∗ I t :∗ I e :∗ Nil)))
  to :: Rep Expr → Expr
  to (Z         (I n :∗ Nil))                 = LInt n
  to (S (Z      (I b :∗ Nil)))                = LBool b
  to (S (S (Z   (I c :∗ I t :∗ I e :∗ Nil)))) = If c t e
```

The complete instance

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

Option 1 – Template Haskell:

```
deriveGeneric ''Expr
```

The complete instance

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

Option 1 – Template Haskell:

```
deriveGeneric ''Expr
```

Option 2:

```
deriving instance GHC.Generics.Generic
instance Generic Expr
```

Translates GHC's generic representation to `generic-sop`'s.
(cf. "Generic Generic Programming" by Magalhães and Löh)

Why this representation?

Representation of Expr in GHC.Generics

```
M1 D D1Expr
    (  M1 C C1_0Expr   (  M1 S NoSelector (K1 R Int))
   :+: M1 C C1_1Expr   (  M1 S NoSelector (K1 R Bool))
   :+: M1 C C1_2Expr   (  M1 S NoSelector (K1 R Expr)
                       :*: M1 S NoSelector (K1 R Expr)
                       :*: M1 S NoSelector (K1 R Expr)
                        )
    )
```

Representation of Expr in GHC.Generics

```
M1 D D1Expr
    (  M1 C C1_0Expr   (  M1 S NoSelector (K1 R Int))
   :+: M1 C C1_1Expr   (  M1 S NoSelector (K1 R Bool))
   :+: M1 C C1_2Expr   (  M1 S NoSelector (K1 R Expr)
                       :*: M1 S NoSelector (K1 R Expr)
                       :*: M1 S NoSelector (K1 R Expr)
                        )
    )
```

- Binary sums and products.
- Arbitrary nesting.
- Noisy metadata everywhere.
- Functions end up making implicit assumptions.

Combinators

Another look at gdef

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

Another look at gdef

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

```
cpure_NP :: (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . (c a) ⇒ f a) → NP f xs
```

Another look at gdef

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

```
cpure_NP :: (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . (c a) ⇒ f a) → NP f xs
```

Somewhat similar to:

```
replicate :: Int → a → [a]
replicate 0 x = []
replicate n x = x : replicate (n − 1) x
```

All

```
cpure_NP :: (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . (c a) ⇒ f a) → NP f xs
```

```
type family All (c :: k → Constraint) (xs :: [k]) :: Constraint
type instance All c '[]       = ()
type instance All c (x ': xs) = (c x, All c xs)
```

All

```
cpure_NP :: (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . (c a) ⇒ f a) → NP f xs
```

```
type family All (c :: k → Constraint) (xs :: [k]) :: Constraint
type instance All c '[]       = ()
type instance All c (x ': xs) = (c x, All c xs)
```

```
All Def '[Int, Bool] ~ (Def Int, (Def Bool, ())) ~ (Def Int, Def Bool)
```

All

```
cpure_NP :: (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . (c a) ⇒ f a) → NP f xs
```

```
type family All (c :: k → Constraint) (xs :: [k]) :: Constraint
type instance All c '[]       = ()
type instance All c (x ': xs) = (c x, All c xs)
```

```
All Def '[Int, Bool] ~ (Def Int, (Def Bool, ())) ~ (Def Int, Def Bool)
```

```
gdef :: (Generic a, All² Def (Code a),
         Code a ~ (xs ': xss), SingI xs) ⇒ a
```

```
type family All² (c :: k → Constraint) (xs :: [[k]]) :: Constraint
```

Creating products

```
cpure_NP :: ∀c xs f . (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . c a ⇒ f a) → NP f xs
cpure_NP p f = case sing :: Sing xs of
                 SNil  → Nil
                 SCons → f :* cpure_NP p f
```

Creating products

```
cpure_NP :: ∀c xs f . (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . c a ⇒ f a) → NP f xs
cpure_NP p f = case sing :: Sing xs of
                 SNil  → Nil
                 SCons → f :* cpure_NP p f
```

```
replicate :: Int → a → [a]
replicate 0 x = []
replicate n x = x : replicate (n − 1) x
```

Example:

```
cpure_NP (Proxy :: Proxy Def) (I def) :: NP I '[Bool, Int]
 = I False :* I 0 :* Nil
```

Many other combinators

Mapping:

```
cliftA_NP :: (All c xs, SingI xs) ⇒ Proxy c
          → (∀a . c a ⇒ f a → g a) → NP f xs → NP g xs
```

Collapsing:

```
collapse_NP :: NP (K a) xs → [a]
```

Sequencing (of applicative effects):

```
sequence_NP :: (SingI xs, Applicative f) ⇒ NP f xs → f (NP I xs)
```

Applying injections:

```
apInjs_NP :: SingI xs ⇒ NP f xs → [NS f xs]
```

Example: a variant of gdef

Let's produce one default value for each constructor (non-recursively):

```
gdefs :: (Generic a, All² Def (Code a)) ⇒ [a]
gdefs = map to (apInjs_POP (cpure_POP (Proxy :: Proxy Def) (I def)))
```

Example: a variant of gdef

Let's produce one default value for each constructor (non-recursively):

```
gdefs :: (Generic a, All² Def (Code a)) ⇒ [a]
gdefs = map to (apInjs_POP (cpure_POP (Proxy :: Proxy Def) (I def)))
```

```
gdef :: (Generic a, All² Def (Code a), Code a ~ (xs ': xss), SingI xs) ⇒ a
gdef = to (Z (cpure_NP (Proxy :: Proxy Def) (I def)))
```

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
gdefs :: [Expr]
 = [LInt 0, LBool False, If (LInt 0) (LInt 0) (LInt 0)]
```

Example: computing the “size” of a value

```
class Size a where
  size :: a → Int

gsize :: (Generic a, All² Size (Code a)) ⇒ a → Int
gsize = (+ 1) . sum . collapse_SOP
      . cliftA_SOP (Proxy :: Proxy Size) (λ(I x) → K (size x)) . from

instance Size Expr where size = gsize
```

Example: computing the “size” of a value

```
class Size a where
  size :: a → Int

gsize :: (Generic a, All² Size (Code a)) ⇒ a → Int
gsize = (+ 1) . sum . collapse_SOP
      . cliftA_SOP (Proxy :: Proxy Size) (λ(I x) → K (size x)) . from

instance Size Expr where size = gsize
```

```
data Expr = LInt  Int
          | LBool Bool
          | If    Expr Expr Expr
```

```
gsize (If (LBool True) (LInt 3) (LInt 5))
 = 7
```

Metadata

Separate metadata

```
type Name = String

data DatatypeInfo :: [[*]] → * where
  ADT     :: Name → NP ConstructorInfo xss → DatatypeInfo xss
  Newtype :: Name → ConstructorInfo '[x] → DatatypeInfo '[ '[x] ]
```

Separate metadata

```
type Name = String

data DatatypeInfo :: [[*]] → * where
  ADT     :: Name → NP ConstructorInfo xss → DatatypeInfo xss
  Newtype :: Name → ConstructorInfo '[x] → DatatypeInfo '[ '[x] ]
```

```
data ConstructorInfo :: [*] → * where
  Constructor :: SingI xs ⇒ Name → ConstructorInfo xs
  Record      :: SingI xs ⇒ Name → NP FieldInfo xs → ConstructorInfo xs

class HasDatatypeInfo a where
  datatypeInfo :: Proxy a → DatatypeInfo (Code a)
```

Separate metadata

```
type Name = String
data DatatypeInfo :: [[*]] → * where
  ADT       :: Name → NP ConstructorInfo xss → DatatypeInfo xss
  Newtype :: Name → ConstructorInfo '[x] → DatatypeInfo '[ '[x] ]
```

```
data ConstructorInfo :: [*] → * where
  Constructor :: SingI xs ⇒ Name → ConstructorInfo xs
  Record :: SingI xs ⇒ Name → NP FieldInfo xs → ConstructorInfo xs
class HasDatatypeInfo a where
  datatypeInfo :: Proxy a → DatatypeInfo (Code a)
```

- Indexed over the code, so it's guaranteed to match the structure.
- Easy to operate on when needed, not in the way otherwise.
- Makes it attractive to define other metadata-like structures.

In the paper

- More detailed explanations, including the implementation of all the important combinators.
- More examples:
  - Computing lenses for record types generically.
  - JSON (de)serialization.
  - Generic validation with user-defined metadata.
- Related work.

Conclusions

You should use this library, because:

- it works (we use it),
- it gives you more structure to work with,
- it encourages high-level code.

Conclusions

You should use this library, because:

- it works (we use it),
- it gives you more structure to work with,
- it encourages high-level code.

Use of type system features:

- GADTs,
- kind polymorphism,
- data kinds (type-level lists),
- type families,
- abstraction over constraints.

Conclusions

You should use this library, because:

- it works (we use it),
- it gives you more structure to work with,
- it encourages high-level code.

Use of type system features:

- GADTs,
- kind polymorphism,
- data kinds (type-level lists),
- type families,
- abstraction over constraints.

Not yet:

- looked at performance (wasn't an issue so far),
- representing higher-kinded types.

On Hackage

`generics-sop` – core library

`basic-sop` – a few simple examples

`lens-sop` – generic lenses (based on `fclabels`)

`json-sop` – generic JSON (de)serialization (based on `aeson`)

Bonus slides

SingI

```
cpure_NP :: (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . (c a) ⇒ f a) → NP f xs
```

To match on a type-level list, we need a term-level handle.

```
class      SingI (a :: k)   where sing :: Sing a
instance SingI '[]          where sing = SNil
instance (SingI x, SingI xs) ⇒
         SingI (x ': xs) where sing = SCons

data family     Sing (a :: k)
data instance Sing (a :: *) = SStar
data instance Sing (xs :: [k]) where
  SNil  :: Sing '[]
  SCons :: (SingI x, SingI xs) ⇒ Sing (x ': xs)
```

Cf. Richard Eisenberg's `singletons` package.

Well-Typed

Proxy

```
cpure_NP :: (All c xs, SingI xs)
         ⇒ Proxy c → (∀a . (c a) ⇒ f a) → NP f xs
```

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

Needed to let the type checker clearly determine `c`.

Representation of lists

```
instance Generic [a] where
  type Code [a] = '[ '[], '[a, [a]]]
  from []       = Z       Nil
  from (x : xs) = S (Z (I x :* I xs :* Nil))
  to (Z       Nil)                    = []
  to (S (Z (I x :* I xs :* Nil))) = x : xs
```

Note that `Generic a` is not required.

Generic validation

```
data Person = Person {name :: String, age :: Int}
```

Generic validation

```
data Person = Person {name :: String, age :: Int}
```

```
class ValidationRules a where
  validationRules :: Proxy a → POP (I ⇾ K Bool) (Code a)
```

Generic validation

```
data Person = Person {name :: String, age :: Int}
```

```
class ValidationRules a where
  validationRules :: Proxy a → POP (I -.> K Bool) (Code a)
```

```
validName :: (I -.> K Bool) String
validName = Fn $ λ(I n) → K (not (null n))

validAge :: (I -.> K Bool) Int
validAge = Fn $ λ(I n) → K (n ⩾ 0)
```

Generic validation

```
data Person = Person {name :: String, age :: Int}
```

```
class ValidationRules a where
  validationRules :: Proxy a → POP (I -.> K Bool) (Code a)
```

```
validName :: (I -.> K Bool) String
validName = Fn $ λ(I n) → K (not (null n))

validAge :: (I -.> K Bool) Int
validAge = Fn $ λ(I n) → K (n ⩾ 0)
```

```
instance ValidationRules Person where
  validationRules _ = (validName :* validAge :* Nil) :* Nil
```

Generic validation

```
data Person = Person {name :: String, age :: Int}
```

```
class ValidationRules a where
  validationRules :: Proxy a → POP (I -.> K Bool) (Code a)
```

```
validName :: (I -.> K Bool) String
validName = Fn $ λ(I n) → K (not (null n))

validAge :: (I -.> K Bool) Int
validAge = Fn $ λ(I n) → K (n ⩾ 0)
```

```
instance ValidationRules Person where
  validationRules _ = (validName :* validAge :* Nil) :* Nil
```

```
validate :: ∀a . (Generic a, ValidationRules a) ⇒ a → Bool
validate =
  and . collapse_SOP . ap_SOP (validationRules (Proxy :: Proxy a)) . from
```