

# GeneralizedNewtypeDeriving is now type-safe!

How **roles** save the day

Richard Eisenberg  
University of Pennsylvania  
[eir@cis.upenn.edu](mailto:eir@cis.upenn.edu)

Haskell Implementors' Workshop  
Sunday, September 22, 2013  
Boston, MA, USA

# GHC 7.6.3 $\Rightarrow$ segfault

```
newtype Age = MkAge Int  
    deriving Frob
```

```
type family Discern a  
type instance Discern Int = Bool  
type instance Discern Age = [Char]
```

```
class Frob a where  
    baz :: a → Discern a  
instance Frob Int where  
    baz = (> 0)
```

```
segfault = head (baz (MkAge 5))
```

# GHC 7.6.3 $\Rightarrow$ segfault

```
newtype Age = MkAge Int  
  deriving Frob
```

```
type family Discern a  
type instance Discern Int = Bool  
type instance Discern Age = [Char]
```

```
class Frob a where  
  baz :: a -> Discern a  
instance Frob Int where  
  baz = (> 0)
```

```
segfault = head (baz (MkAge 5))
```

# Two equalities

**Nominal (N) equality:** two types are the **same**.

- reflexivity “Haskell equality”
- type synonyms “compile-time equality”
- type families
- GADT pattern-matching
- any use of ( $\sim$ )

**Representational (R) equality:** two types have the same **runtime representation**.

- newtypes
- any **nominal** equality “runtime equality”

R-equality is **coarser** than N-equality

# Safety of GND

- GeneralizedNewtypeDeriving (GND) requires representational equality:  
class  $C$   $a$  where ...  
deriving instance  $C$  Age  
 $\Rightarrow$  soundness requires  $(C\ Age \sim_R C\ Int)$
- How do we know if this holds?  
 $\Rightarrow$  depends on the definition of  $C$
- Use a role for the parameter  $a$

# Parameter roles

- All type parameters have a role:

data **Foo** **b** = ...

If **b** is nominal: **Foo** **Age**  $\not\sim_R$  **Foo** **Int**

If **b** is representational: **Foo** **Age**  $\sim_R$  **Foo** **Int**

- **b**'s role says what notion of equality between **Baz** and **Boz** is necessary to prove that **Foo** **Baz** is representationally equal to **Foo** **Boz**.
- A parameter at representational role is more flexible because R-equality is coarser than N-equality

# Type-safe GND

Last parameter of a class has representational role  
 $\Rightarrow$   
GND is type-safe.

# Role inference

- Roles are inferred from a type's definition
- A role is **representational** by default, or **nominal** if a parameter is used in a **nominal** context
- **Nominal** contexts:
  - ✦ Type families
  - ✦ GADT-like parameters
  - ✦ Use with ( $\sim$ )
  - ✦ Other nominal contexts
  - ✦ plus one more...



# Roles examples

class C1 a where m1 :: a → [a]

⇒ a is representational

class C2 a where m2 :: a → Discern a

⇒ a is nominal

data T1 a = MkT1 a

⇒ a is representational

data T2 a where MkT2 :: T2 Bool

⇒ a is nominal

data T3 a = MkT3 (T2 a)

⇒ a is nominal

# Tricky role inference

data Tricky a b = MkTricky (a b)

⇒ a is representational, b is nominal

# Role inference

- Roles are inferred from a type's definition
- A role is **representational** by default, or **nominal** if a parameter is used in a **nominal** context
- **Nominal** contexts:
  - ✦ Type families
  - ✦ GADT-like parameters
  - ✦ Use with ( $\sim$ )
  - ✦ Other nominal contexts
  - ✦ Argument to another type variable

# Role annotations

```
type role Set nominal
```

```
data Set a = ...
```

```
instance Ord Age where ...
```

```
-- inverse of Int's Ord instance
```

```
class HasSet a where mkSet :: Set a
```

```
instance HasSet Int where mkSet = ...
```

```
deriving instance HasSet Age
```

# Role annotations

```
type role Set nominal
```

```
data Set a = ...
```

```
instance Ord Age where ...
```

```
-- inverse of Int's Ord instance
```

```
class HasSet a where mkSet :: Set a
```

```
instance HasSet Int where mkSet = ...
```

Can't make a derived instance of 'HasSet Age'

(even with cunning newtype deriving):

it is not type-safe to use GeneralizedNewtypeDeriving on this class;

the last parameter of 'HasSet' is at role nominal

# Roles break code (1)

- Increase in type safety  $\Rightarrow$  less code compiles
- Case study: Only 2 changes required in GHC
- In `cmm/SMRep.lhs`:

```
newtype StgWord = StgWord Word64
    deriving (IArray UArray, ...)
```

- In `Data.Array.Base`:

```
class IArray a e where
    bounds :: Ix i  $\Rightarrow$  a i e  $\rightarrow$  (i, i)
    ...
```

- Had to manually write wrapper functions

# Roles break code (2)

- Increase in type safety  $\Rightarrow$  less code compiles
- Case study: Only 2 changes required in GHC
- In `utils/UniqFM.lhs`:  

```
newtype UniqFM ele = UFM (IntMap ele)  
    deriving (Traversable, ...)
```
- In `Data.Traversable`:  

```
class (...)  $\Rightarrow$  Traversable t where  
    traverse :: ...  $\rightarrow$  f (t b)  
    ...
```
- Just add `-XDeriveTraversable`

# Roles in GHC

- Most functions that produce a `Coercion` now take a `Role`:

```
mkTyConAppCo :: Role -> TyCon
              -> [Coercion] -> Coercion

dsTcCoercion :: Role -> TcCoercion
             -> (Coercion -> CoreExpr)
             -> DsM CoreExpr
```

- Role conversion is available:

```
maybeSubCo2_maybe :: Role -- desired
                   -> Role -- current
                   -> Coercion -> Maybe Coercion
```

- How to know which role to use? See [ghc/docs/core-spec/core-spec.pdf](http://ghc/docs/core-spec/core-spec.pdf)



# Roles in libraries

- Step 1: Make sure your code compiles
  - ▶ *diagrams* didn't due to potential bug
  - ▶ ... but *lens* did, as did every other library tested
- Step 2: Think about adding role annotations
  - ▶ *Set, Map*
  - ▶ Other abstract types with class-based invariants
  - ▶ Use CPP to keep compatibility with older versions of GHC

# Further reading

- “Generative Type Abstraction and Type-Level Computation” by Weirich, Vytiniotis, Peyton Jones, and Zdancewic (POPL ’11)
- Roles wiki page:  
<http://ghc.haskell.org/trac/ghc/wiki/Roles>
- Roles implementation wiki page:  
<http://ghc.haskell.org/trac/ghc/wiki/RolesImplementation>
- Blog post:  
<http://typesandkinds.wordpress.com/2013/08/15/roles-a-new-feature-of-ghc/>