Safe Zero-cost Coercions for Haskell

Joachim Breitner
Karlsruhe Institute of Technology
breitner@kit.edu

Richard Eisenberg
University of Pennsylvania
eir@cis.upenn.edu

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

Microsoft Research

Karlsruhe Institute for Technology

Tuesday, 2 September, 2014
ICFP, Göteborg, Sweden
Abstraction can be a drag...

newtype HTML = MkH String
  -- MkH is not exported
  -- safety increase over using String
  -- “no runtime overhead”

string :: HTML → String
string (MkH s) = s

stringList :: [HTML] → [String]
stringList hs = map string hs
  -- this no-op takes linear time!
Outline, in brief

I. How we make “zero-cost” abstractions cost nothing, retaining type safety

II. Consequences of our design & other practicalities
A new equivalence relation: $\approx$

\[
\begin{align*}
\text{coerce} & : : \ a \approx b \Rightarrow a \rightarrow b \\
\text{We want:} & \ (a \approx b) \Rightarrow ([a] \approx [b]) \\
\text{So: Use a type class!} & \\
& \text{class } a \approx b \\
& \text{instance } a \approx b \Rightarrow [a] \approx [b] \\
(\approx) & \text{ is spelled Coercible in GHC 7.8}
\end{align*}
\]
coerce must be free!

\[
\text{coerce} :: \ a \approx b \Rightarrow a \rightarrow b
\]

coerce x =

Instances of \(\approx\) must be sound!
Instances of ($\approx$)

Reflexivity:

\texttt{instance } a \approx a

No symmetry or transitivity:
we need syntax-directed solving

Symmetry and transitivity are \texttt{admissible}
Instances of \((\approx)\)

From newtype declarations:

\[
\text{newtype } \text{HTML} = \text{MkH String} \quad \Rightarrow \\
\text{instance } a \approx \text{String} \Rightarrow a \approx \text{HTML} \\
\text{instance } \text{String} \approx b \Rightarrow \text{HTML} \approx b
\]

Assume newtype \(\text{ValidHTML} = \text{MkV HTML}\)

Can derive \((\text{String} \approx \text{ValidHTML})\):

\[
\text{String} \approx \text{String} \\
\Rightarrow \text{String} \approx \text{HTML} \\
\Rightarrow \text{String} \approx \text{ValidHTML}
\]
Instances of $(\approx)$

From `data` declarations:

```haskell
data Maybe a = Nothing | Just a

instance a ≈ b ⇒ Maybe a ≈ Maybe b
```

Lifting instances also made for `newtypes`

Can derive `(Maybe HTML ≈ Maybe String)`:

```haskell
String ≈ String
⇒ HTML ≈ String
⇒ Maybe HTML ≈ Maybe String
```
But that’s too permissive!

```haskell
type family F a

type instance F String = Int

type instance F HTML = Bool → Bool

newtype UhOh a = MkUO (F a)

Can derive (Int ≈ Bool → Bool): (!!!)

⇒ ... ⇒ UhOh String ≈ UhOh HTML  lifting

⇒ ... ⇒ F String ≈ F HTML  unwrapping

⇒ Int ≈ Bool → Bool
```
But that's too permissive!

Can derive:

\[
\int \approx \text{Bool} \rightarrow \text{Bool}
\]

⇒

\[
\text{String} \approx \text{UhOh HTML}
\]

⇒

\[
\text{F String} \approx \text{F HTML}
\]

⇒

\[
\int \approx \text{Bool} \rightarrow \text{Bool}
\]

= lifting unwrapping image:

Jim Urquhart/Reuters

(newtype)

\[
\text{UhOh} a = \text{MkUO} (\text{F} a)
\]
A tale of two equalities

\[ (~) \]
nominal

compile time
equal in Haskell code
automatic conversion
finer
\[ (x \sim y) \]
\[ (x \sim y) \]
⇒

\[ (\approx) \]
representational
run time
equal to code generator
manual conversion
coarser
\[ (x \approx y) \]
\[ (x \approx y) \]
⇔
A tale of two equalities

Type families, GADTs, class instances, etc. can distinguish a newtype and its representation.

```haskell
type instance F String  = Int
type instance F HTML   = Bool -> Bool
```

Does not respect $(\approx)$
Roles

We must differentiate between

data Maybe a  
= Nothing 
| Just a  

newtype UhOh a  
= MkUO (F a)

Answer: assign roles to type parameters

Adaptation of ideas in previous work [1]:

• Simpler -- doesn’t require a new kind system
• Less expressive -- some higher-order types excluded
• More flexible -- roles aren’t in kinds

[1]: Weirich, Vytiniotis, Peyton Jones, Zdancewic. Generative type abstraction and type-level computation. POPL ’11
Roles

Three roles:
• Nominal \( (n) \)
• Representational \( (r) \)
• Phantom \( (p) \)

\( n \) parameter is unchanged
\( r_1 \) and \( r_2 \) must be representationally equal
No relationship between \( p_1 \) and \( p_2 \)

Examples:

\( \text{UhOh} \)

\( \text{Maybe, [], Either} \)

Data Proxy \( a = P \)

Instance \( (\text{UhOh } n) \approx (\text{UhOh } n) \) -- redundant

Instance \( r_1 \approx r_2 \Rightarrow (\text{Maybe } r_1) \approx (\text{Maybe } r_2) \)

Instance \( \text{Proxy } p_1 \approx \text{Proxy } p_2 \)
Role Inference

Goal: Determine the most permissive yet safe role for type parameters

\[ P > R > N \]

\( (> ) \equiv \text{“more permissive than”} \)

Algorithm: Find fixed point of propagating role restrictions

Nominal roots: type families, \((\sim)\), GADTs, ...
Representational roots: \((\rightarrow)\), ...
Type Safety

Proved progress and preservation using GHC’s typed intermediate language, System FC.
instance Num Int where ...
newtype Age = MkAge Int
deriving Num

Num Age instance built from coerced methods of Num Int instance.

GeneralizedNewtypeDeriving (GND) is a long-standing feature of GHC, now safely reimplemented in terms of coerce.
Abstraction

Q: If `HTML ≈ String`, what happens to safety?

A: Allow newtype (un)wrapping instances only when constructor is in scope
Abstraction

• A Map $k \rightarrow v$ maps keys $k$ to values $v$
• Keys are ordered by $k$’s Ord instance
• Map is abstract -- its constructor is not in scope

Q: Should $\text{Map Int String} \approx \text{Map Int HTML}$?
A: Yes!

Q: Should $\text{Map String Int} \approx \text{Map HTML Int}$?
A: No -- What if String’s Ord is not HTML’s Ord?
Abstraction

data Map k v = MkMap [(k, v)]

The programmer should specify the roles:
type role Map nominal representational
The Default Debate

Preserve abstraction! Make roles default to nominal!

Be backward compatible! Allow GeneralizedNewtypeDeriving!

GHC 7.8 infers the most permissive roles.
Roles in the Wild

• Roles were included in the GHC dev build on Aug. 2, 2013.
• On Sept. 30, Bryan O’Sullivan did a study, trying to compile all of Hackage\(^1\)
• 3,234 packages compiled with GHC 7.6.3
• Only 4 failed due to compile due to role restrictions around GND
• 3 of these 4 were legitimate bugs
• 1 was due to conservativity of roles

Trouble on the Horizon?

Proposed new Monad class:

```haskell
class (...) ⇒ Monad m where
...
join :: forall a. m (m a) → m a
```

Imagine

```haskell
newtype Restr m a = MkR (m a)
deriving Monad
```

Given: `Restr m a ≈ m a`

Wanted: `Restr m (Restr m a) ≈ m (m a)`
Trouble on the Horizon?

Given: \( \text{Restr } m \ a \approx m \ a \)

Wanted: \( \text{Restr } m \ (\text{Restr } m \ a) \approx m \ (m \ a) \)

\[ \uparrow \]

\[ m \ (\text{Restr } m \ a) \approx m \ (m \ a) \]

\[ \uparrow \]

\[ ??????? \]

\( m \)'s parameter's role might be nominal, so we're stuck!
Trouble on the Horizon?

Proposed solution: Track variable-parameter roles via typeclasses.

See https://ghc.haskell.org/trac/ghc/wiki/Roles2
Added Flexibility

Roles do not appear in a variable’s kind.

class Functor (f :: ★ → ★) where ... 
instance Functor Maybe where ... 
instance Functor UhOh where ... 

This would not work with the previous formulation of roles.
Conclusion

• Allowed for an efficient, safe way to make zero-cost abstractions truly free.

• Straightforward interface: \((\cong)\)/Coercible

• Implemented and released in GHC 7.8

• Explored interaction between type abstraction and other type features; these issues exist in other languages too (e.g. OCaml’s variance annotations)