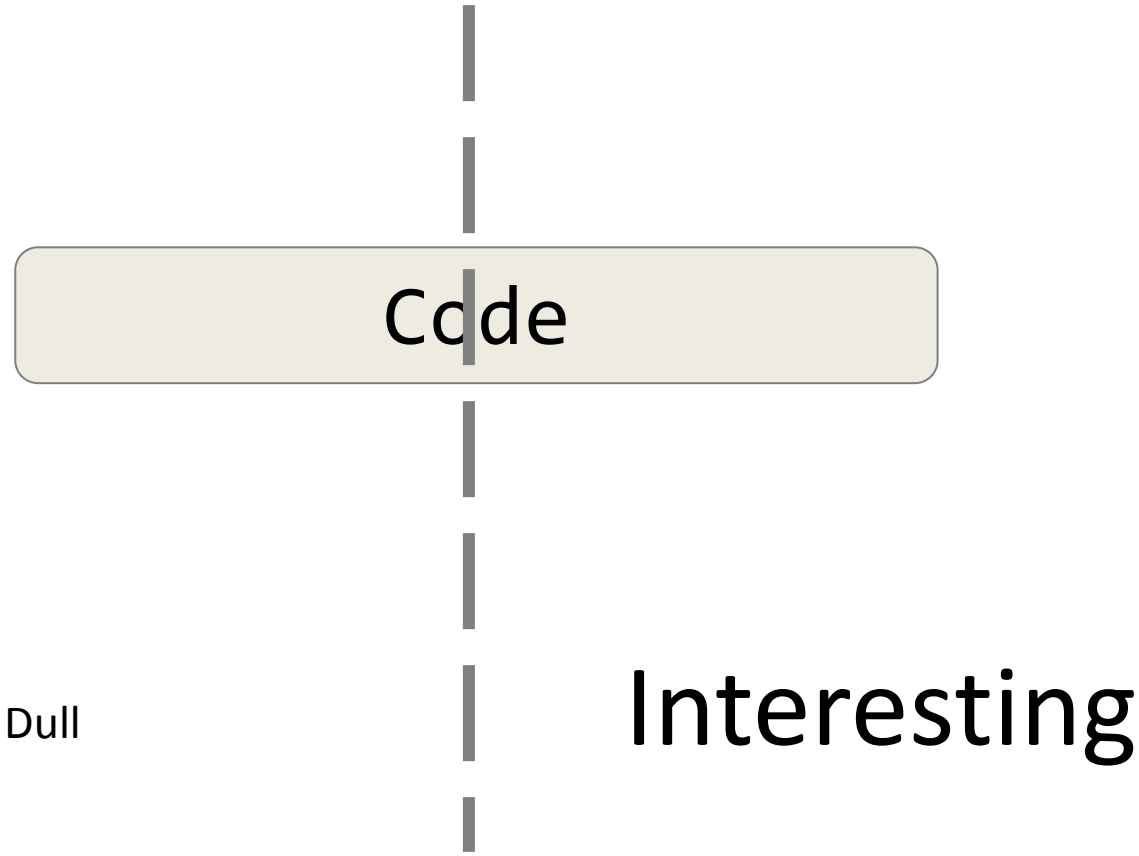
Everyone should use a generics library!

Writing HLint with Uniplate

Neil Mitchell

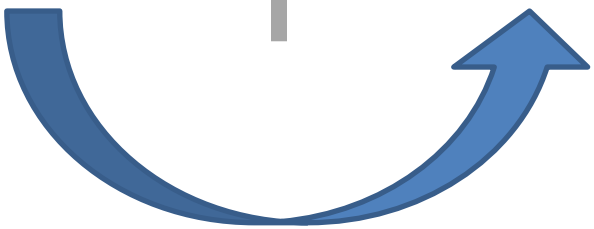

Code
Dull
Interesting

HLint

- Tool to make your code better
- Suggest improvements

Uniplate

- Library for short, concise, flexible code
- A “generics” library



Uses

cabal install hlint

HLint

```
Main.hs:1:1: Error: Unused LANGUAGE pragma
Found: {-# LANGUAGE ViewPatterns #-}
Why not remove it.
```

```
Main.hs:4:1: Warning: Use foldr
Found:
  looper (x : xs) = x <!> looper xs
  looper [] = []
Why not:
  looper xs = foldr (<!>) [] xs
```

```
Main.hs:11:21: Error: Use minimumBy
Found:   head $ sortBy f xs
Why not: minimumBy f xs
```

A new hint

> “GHC used to support the flag -XRecursiveDo, which enabled the keyword mdo, but this is now deprecated.
>
> Instead of mdo { Q; e }, write do { rec Q; e }.”

From GHC 7.0 manual
(by GHC 7.6 mdo was undeprecated again)

The Plan

Find

`mdo`

Replace with

`do`

```
mdo
    x <- foo y
    y <- bar x
    return (x+y)
```

```
do
    rec x <- foo y
        y <- bar x
    return (x+y)
```

cabal install haskell-src-exts

The Code

```
mdo {}            ==> do {}
mdo {x1; x2; x3} ==> do {rec {x1; x2}; x3}
```

```
removeMDo :: Exp -> Exp
removeMDo (MDo []) = Do []
removeMDo (MDo xs) = Do [RecStmt (init xs), last xs]
```

- Clear and concise ☺

The Pieces

```
removeMDo :: Exp -> Exp
```

```
parseFile :: FilePath -> IO (ParseResult Module)
```

```
onModule :: Module -> Module
onModule = ... removeMDo ...
```

- How do we join the pieces?

Putting the pieces together

```
data Module
```

A complete Haskell source module.

Constructors

```
Module SrcLoc ModuleName [ModulePragma] (Maybe WarningText)
       (Maybe [ExportSpec]) [ImportDecl] [Decl]
```

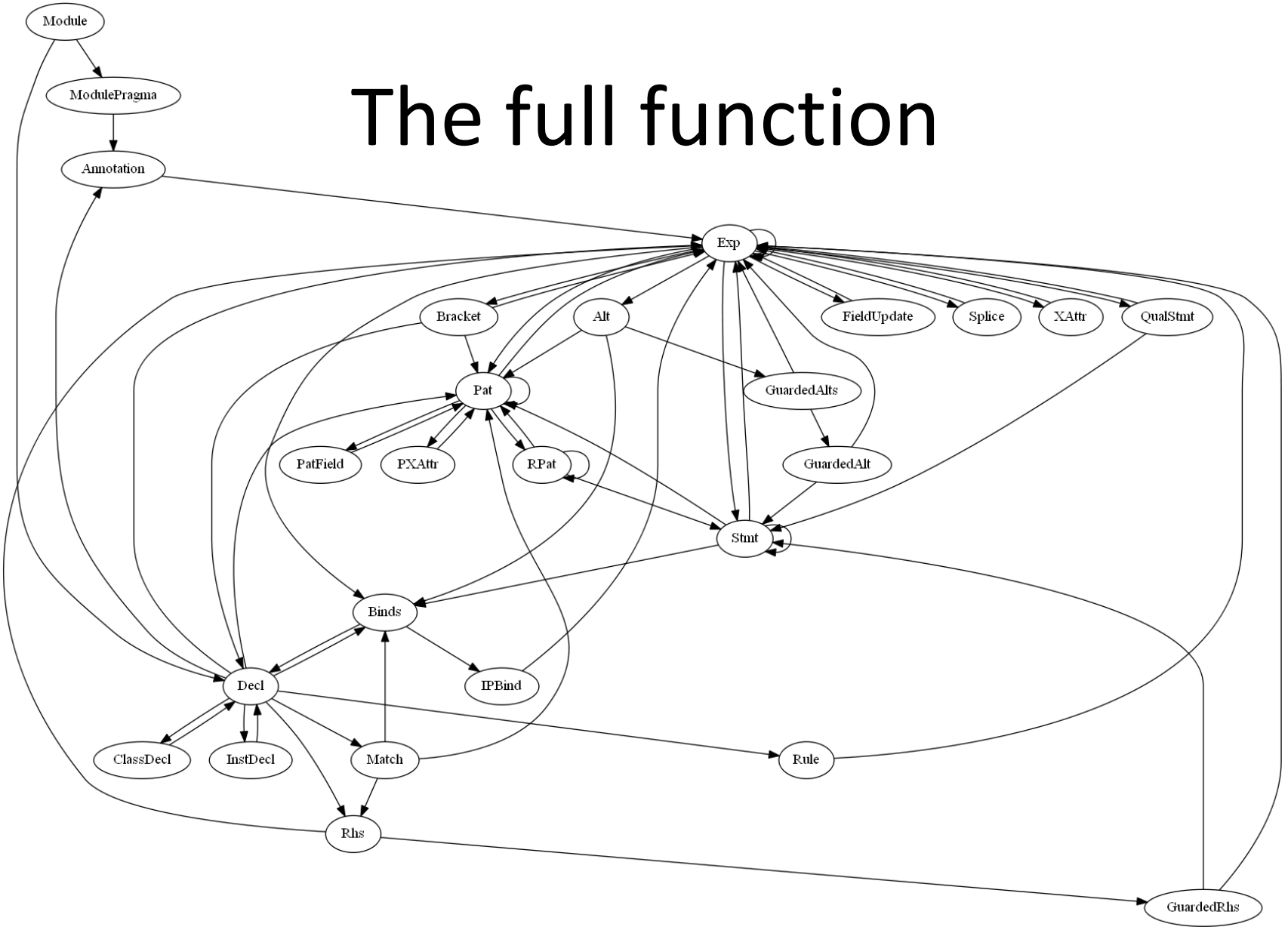
```
onModule :: Module -> Module
onModule (Module x1 x2 x3 x4 x5 x6 x7) =
  Module x1 x2 x3 x4 x5 x6 (map onDecl x7)
```

Putting the pieces together

```
DataFamDecl SrcLoc Context Name [TyVarBind] (Maybe Kind)
TypeInsDecl SrcLoc Type Type
DataInsDecl SrcLoc DataOrNew Type [QualConDecl] [Deriving]
```

```
onDecl :: Decl -> Decl
onDecl (ClassDecl x1 x2 x3 x4 x5 x6) =
  ClassDecl x1 x2 x3 x4 x5 (map onClassDecl x6)
onDecl (InstDecl x1 x2 x3 x4 x5) =
  InstDecl x1 x2 x3 x4 (map onInstDecl x6)
onDecl (SpliceDecl x1 x2) = SpliceDecl x1 (onExp x2)
onDecl (FunBind x1) = FunBind (map onMatch x1)
onDecl (PatBind x1 x2 x3 x4 x5) =
  PatBind x1 x2 x3 (onRhs x4) (onBinds x5)
onDecl (RulePragmaDecl x1 x2) = RulePragmaDecl x1 (map onRule x2)
onDecl (AnnPragma x1 x2) = AnnPragma x1 (onAnnotation x2)
onDecl x = x
```

The full function



Did you spot the mistake on the previous slides?

26 types, 107 constructors = 159 lines

cabal install uniplate

The good news!

```
import Data.Generics.Uniplate.Data


removeMDo :: Exp -> Exp
removeMDo (MDo []) = Do []
removeMDo (MDo xs) = Do [RecStmt (init xs)] (last xs)
removeMDo x = x


onModule :: Module -> Module
onModule = transformBi removeMDo
```

Generics express pattern + exceptions

- Concise
- Fewer errors
- Reusable
- Robust to library changes

Uniplate Generic Patterns

	Query	Transform
Deep	`universe` (find all)	`transform` (global replacement)
Shallow	`children` (reduce to a value)	`descend` (top-down with control)

```
transform :: (on -> on) -> on -> on
```

Bottom-up traversal – transform workhorse (15 in hlint)

```
lessParen :: Exp -> Exp
lessParen = transform f
  where f (Paren (Paren x)) = Paren x
        f (Paren (List x))  = List x
        f x                 = x
```

```
transformBi  :: (to ->   to) -> from -> from
transformM   :: (on -> m on) -> on   -> m on
transformBiM :: (to -> m to) -> from -> m from
```

Assuming: `(Biplate from to, Uniplate on, Monad m) => ...`

```
universe :: on -> [on]
```

Find all elements – querying workhorse (24 in hlint)

```
redundantExtension :: Module -> Bool
redundantExtension m =
    viewPats == 0 && “ViewPatterns” `elem` exts
  where
    viewPats = length [() | PViewPat{} <- universeBi m]
    exts = [prettyPrint x
        | LanguagePragma _ xs <- universeBi m, x <- xs]
```

```
universeBi :: from -> [to]
```

```
descend :: (on -> on) -> on -> on
```

One-level traversal – used for top-down with control/context

(11 in hlint)

```
eval :: Exp -> Exp
eval x = case x of
    Lambda{} -> x
    If c t f | prettyPrint c == “True”  -> eval t
             | prettyPrint c == “False” -> eval f
    x -> descend eval x
```

```
descendBi  :: (to ->   to) -> from -> from
descendM   :: (on -> m on) -> on   -> m on
descendBiM :: (to -> m to) -> from -> m from
```

```
children :: on -> [on]
```

One-level children – reduction defaults (20 in hlint)

```
freeVars :: Exp -> [String]
freeVars (Var x) = [prettyPrint x]
freeVars (Lambda _ x bod) = freeVars bod \\ boundVars x
...
freeVars x = nub $ concatMap freeVars $ children x
```

```
childrenBi :: from -> [to]
```

Pick the pattern!

`transform`
(global replacement)

`universe`
(find all)

`descend`
(top-down with control)

`children`
(reduce to a value)

Update the copyright year

Do I use any “magic” constants?

Migrate to a new version of GHC

Cyclometric complexity

What language pragma do I use most?

Fix spelling mistakes in code

CSE long strings

Inline `id`

Recursive type +
> 3 ctors

- Any abstract syntax tree
- Any compiler
- …

Boilerplate (per type)

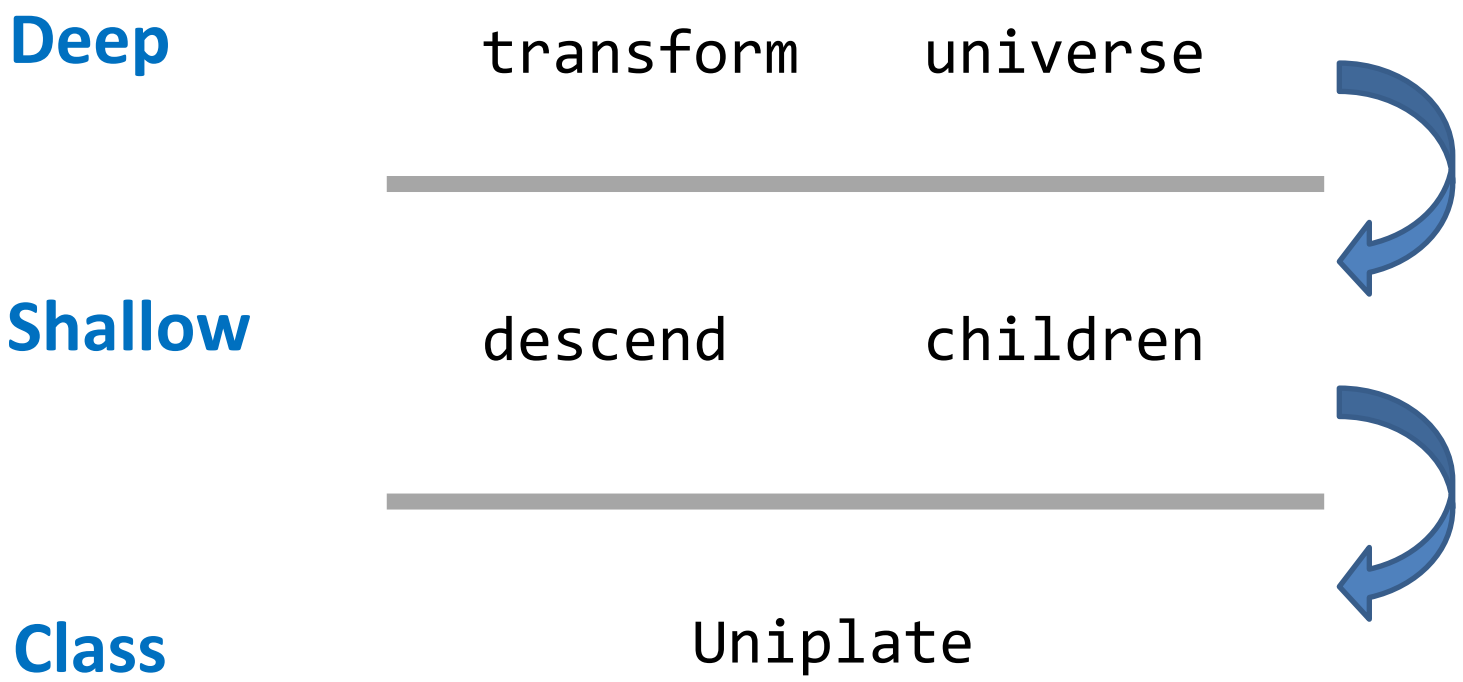
```
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Data
import Data.Generics.Uniplate.Data

data MyType = MyType ... deriving (Data,Typeable)
```

- Now `MyType` is Uniplate ready

Implementing Uniplate (in 2 slides)



Note: This is the original uniplate implementation, which is simpler, and 90% as good as the real one

Deep in terms of Shallow

```
universe x = x : concatMap universe (children x)
transform f = f . descend (transform f)
```

Shallow in terms of class

```
class Uniplate on where
    uniplate :: on -> ([on], [on] -> on)
```

```
instance Uniplate Exp where
    uniplate (App x y) = ([x,y], \[x,y] -> App x y)
    uniplate (List xs) = (xs, \xs -> List xs)
    ...
```

```
children = fst . uniplate

descend f x = gen $ map f cs
    where (cs, gen) = uniplate x
```

Uniplate in HLint

- I *never* have cases for each constructor
  - Decl has 27, Exp has 45, Pat has 23 ...
- A generics library is essential
- Why Uniplate?
  - Simple
  - Concise
  - Performant
  - Author colocation

Why not more Uniplate?

- HLint has > 400 hints, but < 100 Uniplate uses

- Two reasons:
  - Defining new reusable patterns
  - HLint “hint” language

New Patterns

```
moduleDecls :: Module -> [Decl]
moduleDecls = childrenBi


declHint :: (Decl -> [Idea]) -> (Module -> [Idea])
declHint f = concatMap f . moduleDecls


universeParent ::
    Biplate from to => from -> [(Maybe (Int, to), to)]
universeParent xs = concat
    [(Nothing, x) : f x | x <- childrenBi xs]
  where f p = concat [ (Just (i,p), c) : f c
                     | (i,c) <- zip [0..] $ children p]
```

New Views

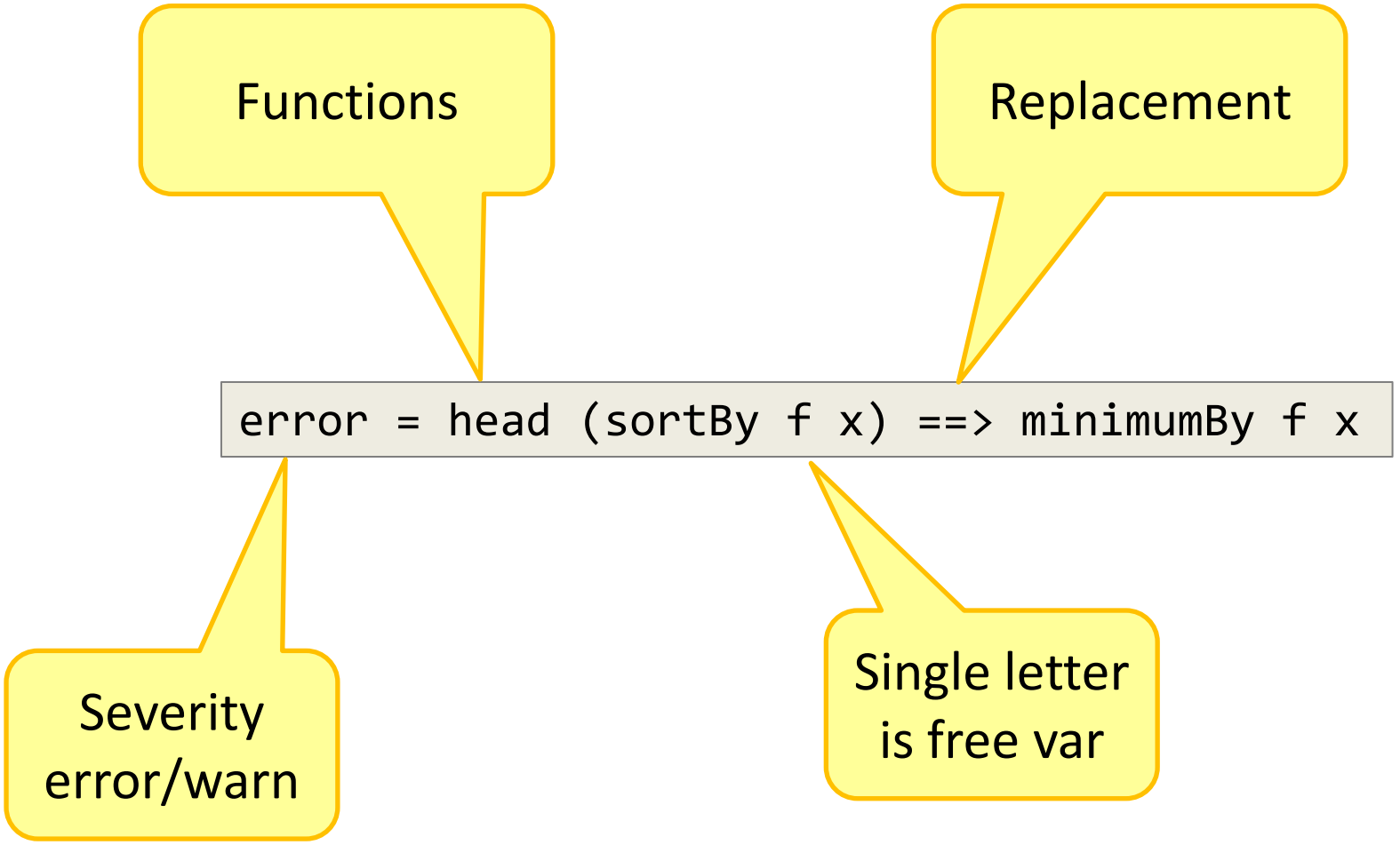
```
fromApps :: Exp -> [Exp]
fromApps (App x y) = fromApps x ++ [y]
fromApps x = [x]


toApps :: [Exp] -> Exp
toApps = foldl1 App


descendApps :: (Exp -> Exp) -> Exp -> Exp
descendApps f (App x y) = App (descendApps f x) (f y)
descendApps f x = descend f x


transformApps :: (Exp -> Exp) -> Exp -> Exp
transformApps = f . descendApps (transformApps f)
```

Hint language



332 hints at last count

Care-free matching

```
error = head (sortBy f x) ==> minimumBy f x
error = head . sortBy f   ==> minimumBy f
```

```
a         ==> b
(a)       ==> a
a $ b     ==> a (b)
a . b c   ==> a (b c)
```

- Heuristic based module name resolution
- Special treatment of “_” patterns

Implementing Matches

```
unify :: Exp -> Exp -> Maybe [(String,Exp)]
unify (Var x) y = Just [(x,y)]
unify x y | ctor x /= ctor y = Nothing
unify x y = fmap concat $ sequence $
    zipWith unify (children x) (children y)


ctor :: Exp -> String
ctor = takeWhile (not . isSpace) . show
```

- Missing lots of ugly details (brackets, name resolution, $, Pat...)
- Unroll a few common cases (App)
- zipWith/ctor implemented using Data.Data for performance and to hit both Exp and Pat

Real(er) Implementation

```
unify :: Data a => a -> a -> Maybe [(String,Exp_)]
unify x y
  | Just x <- cast x = unifyExp x $ unsafeCoerce y
  | Just x <- cast x = unifyPat x $ unsafeCoerce y
  | otherwise = fmap concat . sequence =<< gzip unify x y
```

```
data Box = forall a . Data a => Box a
gzip :: Data a => (forall b . Data b => b -> b -> c)
                -> a -> a -> Maybe [c]
gzip f x y = if toConstr x /= toConstr y then Nothing else
    Just $ zipWith op (gmapQ Box x) (gmapQ Box y)
  where op (Box x) (Box y) = f x (unsafeCoerce y)
```

Correctness

```
warn = reverse (reverse x) = x
    where note = IncreasesLaziness
```

lemma "$reverse \cdot (reverse \cdot xs) \sqsubseteq xs$"
proof $(induction\ xs)$
  case $(Cons\ x\ xs)$
  have "$reverse \cdot (reverse \cdot (x{:}xs)) = reverse \cdot (reverse \cdot xs +\!\!+ [x])$" **by** $simp$
  also have "$\ldots \sqsubseteq reverse \cdot [x] +\!\!+ reverse \cdot (reverse \cdot xs)$" **by** $(rule\ rev\_append)$
  also have "$\ldots = x : reverse \cdot (reverse \cdot xs)$" **by** $simp$
  also have "$\ldots \sqsubseteq x : xs$" **by** $(simp\ add{:}\ Cons.IH)$
  finally show $?case$.
qed $simp\_all$

143 hints have been proven correct

"Cerified HLints with Isabelle/HOLCF-Prelude"

Haskell and Rewriting Techniques 2013

cabal install nsis

Uniplate in NSIS

```
data NSIS
  = Labelled Label
  | Goto Label
  | StrNull Var Label
  | Fun Name [NSIS]
  | ...
```

- NSIS = Windows installer
- Goto with fixed number of string registers
- Haskell EDSL to restore sanity

```
strnull $1 is_null
goto not_null
```

```
strnull $1 is_empty
goto not_empty
```

```
:is_null
goto is_empty
```

```
:not_null
:not_empty
...
```

Elimination

```
data NSIS
  = Labelled Label
  | Goto Label
  | StrNull Var Label
  | Fun Name [NSIS]
  | ...
```

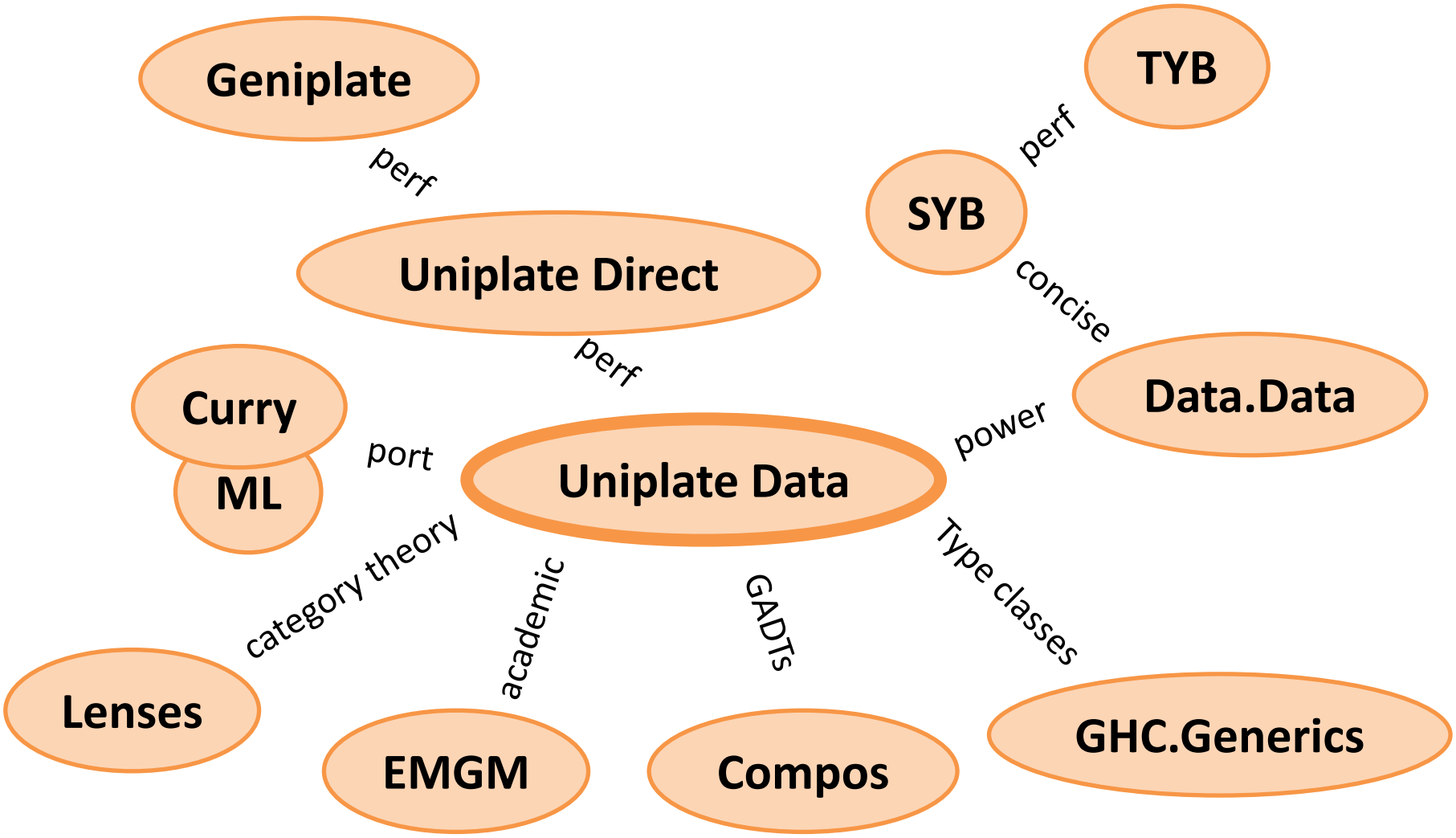
```
elimLabeledGoto :: [NSIS] -> [NSIS]
elimLabeledGoto o = transformBi f o
  where
    f (Labelled x) = Labelled x
    f x | not $ null $ children x = x
    f x = descendBi moveBounce x

    moveBounce l = fromMaybe l $ lookup l bounce

    bounce = flip concatMap (universe o) $ \x ->
        case x of
            Labelled l:Goto l2:_ -> [(l,l2)]
            Labelled l:Labelled l2:_ -> [(l,l2)]
            _ -> []
```

```
cabal install geniplate syb TYB lenses emgm
```

Beyond Uniplate



cabal install uniplate

Summary

```
import Data.Generics.Uniplate.Data
```

	Query	Transform
Deep	`universe` (find all)	`transform` (global replacement)
Shallow	`children` (reduce to a value)	`descend` (top-down with control)


I compilrd
but haz no genricz
ICANHASCHEEZBURGER.COM