GHC/Haskell Language Extensions: A Digest

Allele Dev (@queertypes)

February 19, 2015

Hi, This is Me

- Github: cabrera
- Gitlab: cpp.cabrera
- Twitter: @queertypes
- General Blog: `https://blog.cppcabrera.com/`
- Gamedev Blog: `https://applicative-games.cppcabrera.com/`

Overview

- Extensions as a Haskell User

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes
- Small, Easy, and Quick Extensions

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes
- Small, Easy, and Quick Extensions
- Diving In

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes
- Small, Easy, and Quick Extensions
- Diving In
  - GADTs

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes
- Small, Easy, and Quick Extensions
- Diving In
  - GADTs
  - Existential Quantification

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes
- Small, Easy, and Quick Extensions
- Diving In
    - GADTs
    - Existential Quantification
    - Generalized Newtype Deriving

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes
- Small, Easy, and Quick Extensions
- Diving In
    - GADTs
    - Existential Quantification
    - Generalized Newtype Deriving
    - Empty Data Declarations

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes
- Small, Easy, and Quick Extensions
- Diving In
    - GADTs
    - Existential Quantification
    - Generalized Newtype Deriving
    - Empty Data Declarations
    - Scoped Type Variables

Overview

- Extensions as a Haskell User
- Extension Categories, Broadly
- What I Won't Cover
- Haskell 2010 in 5 Minutes
- Small, Easy, and Quick Extensions
- Diving In
  - GADTs
  - Existential Quantification
  - Generalized Newtype Deriving
  - Empty Data Declarations
  - Scoped Type Variables
- Resources

Extensions as a Haskell User

- How do you enable extensions?

Example

A Haskell program.

```haskell
module Main where

main = print "Hi"
```

Example

A (Haskell + OverloadedStrings) program.

```haskell
{-# LANGUAGE OverloadedStrings #-}
module Main where

main = print "Hi"
```

Example

A (Haskell + OverloadedStrings + GADTs) program.

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE GADTs #-}
module Main where

main = print "Hi"
```

Enabling Extensions

- Place them at the top of a file (cannonical, common)

Enabling Extensions

- Place them at the top of a file (cannonical, common)
- Place them in a cabal file: `extensions: <name>`

Enabling Extensions

- Place them at the top of a file (cannonical, common)
- Place them in a cabal file: `extensions: <name>`
    - `NoImplicitPrelude` makes a lot of sense here

Enabling Extensions

- Place them at the top of a file (cannonical, common)
- Place them in a cabal file: `extensions: <name>`
  - `NoImplicitPrelude` makes a lot of sense here
- Specify them on the command line/build: -X

Extension Categories

- Not in a mathematical sense

Extension Categories

- Not in a mathematical sense
  - Extensions are just Functors between all of the Hask Categories, perhaps?

Extension Categories

- Not in a mathematical sense
  - Extensions are just Functors between all of the Hask Categories, perhaps?
- Categorized by what part of the language they change

Extension Categories

- Not in a mathematical sense
  - Extensions are just Functors between all of the Hask Categories, perhaps?
- Categorized by what part of the language they change
- Just one categorization: deviates a little from GHC User Guide

Extension Categories

- Syntactic: add sugar, maintain semantics

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match
- Data construction: allow for more to be expressed at type definition time

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match
- Data construction: allow for more to be expressed at type definition time
- Derivation: allows for more to be derived

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match
- Data construction: allow for more to be expressed at type definition time
- Derivation: allows for more to be derived
- Typeclasses: model more using typeclasses and instances

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match
- Data construction: allow for more to be expressed at type definition time
- Derivation: allows for more to be derived
- Typeclasses: model more using typeclasses and instances
- Generic Programming

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match
- Data construction: allow for more to be expressed at type definition time
- Derivation: allows for more to be derived
- Typeclasses: model more using typeclasses and instances
- Generic Programming
- Evaluation modifiers

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match
- Data construction: allow for more to be expressed at type definition time
- Derivation: allows for more to be derived
- Typeclasses: model more using typeclasses and instances
- Generic Programming
- Evaluation modifiers
- RTS Additions

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match
- Data construction: allow for more to be expressed at type definition time
- Derivation: allows for more to be derived
- Typeclasses: model more using typeclasses and instances
- Generic Programming
- Evaluation modifiers
- RTS Additions
- Signatures and Inference

Extension Categories

- Syntactic: add sugar, maintain semantics
  - Record sugar, new literals, more ways to pattern match
- Data construction: allow for more to be expressed at type definition time
- Derivation: allows for more to be derived
- Typeclasses: model more using typeclasses and instances
- Generic Programming
- Evaluation modifiers
- RTS Additions
- Signatures and Inference
- Dependent Types

What I Cover

- Several small, simple, sugary extensions

What I Cover

- Several small, simple, sugary extensions
- GADTs

What I Cover

- Several small, simple, sugary extensions
- GADTs
- Existential Quantification

What I Cover

- Several small, simple, sugary extensions
- GADTs
- Existential Quantification
- Generalized Newtype Deriving

What I Cover

- Several small, simple, sugary extensions
- GADTs
- Existential Quantification
- Generalized Newtype Deriving
- Empty Data Declarations

What I Cover

- Several small, simple, sugary extensions
- GADTs
- Existential Quantification
- Generalized Newtype Deriving
- Empty Data Declarations
- Scoped Type Variables

Haskell 2010

- What does Haskell 2010 *the language* look like?

Haskell 2010

- What does Haskell 2010 *the language* look like?
- We build intuition from here to see how extensions change the language

Haskell 2010: Module Exports

```haskell
module AllTheThings (
  makeOlder, Person(..)
) where
```

Haskell 2010: Module Imports

```haskell
-- import everything from Data.List
import Data.List

-- import everything from Prelude except foldr
import Prelude hiding (foldr)

-- import everything from ByteString, but must be
-- prefixed with "B." to access
import qualified Data.ByteString as B

-- import only foldr from Data.Foldable
import Data.Foldable (foldr)
```

Haskell 2010: Type Definition

```haskell
-- generate new types treated distinctly
newtype Age = Age Int

-- type alias; treated identically
type Contents = Contents String

-- sum types; "one-of" types; "or" types
data Color = Red | Green | Blue

-- product types: "each-of" types; "and" types
data Box = Box Age Contents

-- recursive types and polymorphism
data Tree a = Leaf a | Branch (Tree a) (Tree a)

-- record types
data Person = Person { age :: Age
```

Haskell 2010: Deriving

- Six classes that can be automatically implemented by compiler
  - Read, Show, Eq, Ord, Enum, Bounded

```
data Color = Red | Green | Blue deriving (Show, Eq)
> Red == Green
False
> Green
Green
```

Haskell 2010: Typeclasses

- Interface mechanism that allows for overloading based on type

```haskell
class Render a where
  render :: a -> String

instance Render Color where
  render Green = "Green"
  render Red = "Red"
  render Blue = "Blue"
```

Haskell 2010: Functions and Type Signatures

```haskell
-- name :: (type ->)* -> return_type
f :: Int -> String

--    v typeclass constraints
g :: Show a => a -> String
```

Haskell 2010: Functions and Pattern Matching

```haskell
f' :: Color -> Int
f' Red = 1
f' Green = 2
f' Blue = 3

g' :: Color -> Int
g' color = case color of
  Red -> 1
  Green -> 2
  Blue -> 3
```

Haskell 2010: Functions and Local Bindings

```
-- let: local bindings prior to function definition
-- where: local bindings after function definition
h :: Int -> Int
h x = let x' = x + 10 in
  multiply x' 7
  where multiply x'' y = x'' * y
```

Haskell 2010: Records and Pattern Matching

```haskell
-- match all fields
makeOlder :: Person -> Person
makeOlder (Person {age= Age(newAge), name=name}) =
  Person {age=Age (newAge+1), name=name}

-- match one field
peekOlder :: Person -> Age
peekOlder (Person {age= Age(age)}) = Age (age+1)
```

Haskell 2010: Functions and Guards

```haskell
newtype HpPct = HpPct Int deriving Show
data State =
  Good | Okay | NotSoGood | Bad | Nope
  deriving Show

-- guards; boolean matching convenience
hpState :: HpPct -> State
hpState (HpPct x)
  | x == 100 = Good
  | x > 75 = Okay
  | x > 50 = NotSoGood
  | x > 25 = Bad
  | otherwise = Nope
```

Haskell 2010: Partial Application

```
f :: Int -> Int -> Int
f x y = x + y

> :t f 1
f 1 :: Int -> Int
> (f 1) 2
3
```

Haskell 2010: Sections

```
-- partial-application sugar
> map (+1) [1,2,3,4]
[2,3,4,5]

-- without sugar
> map (\x -> x + 1) [1,2,3,4]
[2,3,4,5]
```

```haskell
-- f is polymorphic over a; for all a, f must work
f :: a -> a
f x = x

-- swap is polymorphic over a and b
swap :: (a,b) -> (b,a)
swap (x,y) = (y,x)

-- reverse can operate on homogeneous lists
reverse :: [a] -> [a]
```

Haskell 2010: Higher Kinds

- Kinds: the types of types

```
data Maybe a = Just a | Nothing
data Either a b = Left a | Right b

-- Maybe requires one type to become a usable type
> :kind Maybe
Maybe :: * -> *

-- Either requires two type
> :kind Either
Either :: * -> * -> *
```

Haskell 2010: do-notation

- do-notation: sugar over (>>=) for Monad instances

```
main = do
  contents <- getLine
  print contents

main = getLine >>= (\contents -> print contents)
```

Haskell 2010: List Comprehensions

```
> [x | x <- [1..10], x > 4]
[5,6,7,8,9,10]
```

Haskell 2010: Review

- Modules
- Type definition
- Deriving
- Typeclasses
- Functions
- Type signatures
- Pattern matching
- Records
- Guards
- Partial application and section sugar
- Parametric polymorphism
- Do-notation
- List comprehensions
- A little more

Simple Extensions

- On to extensions now - simple extensions

Simple Extensions

- On to extensions now - simple extensions
- These are extensions that are simpler and take less to understand

Magic Hash

- Key idea: enables the use of '#' as a suffix in the names of things
- That's it
- So why does this matter?

Magic Hash

- GHC primitives are all exposed as '#' suffixed types
- To access those unboxed types, you'll need this extension
- See: GHC.Exts

Binary Literals (7.10)

- New in GHC 7.10: binary literals, e.g., `0b0010`
- Interpreted as `fromInteger <literal_as_int>`

Pattern Guards

- Generalized guards: allows pattern matching rather than just `Bool` predicates

```haskell
f :: Maybe Int -> Maybe Int -> Maybe Int
f x y
  | Just x' <- x
  , Just y' <- y = Just $ x' + y'
  | otherwise = Nothing
```

Tuple Sections

- Section syntax extended to tuples
- Partial application over tuple construction

```
> :t (,1,,,'a')
(,1,,,'a') :: Num t1 => t -> t2 -> t3 ->
                           (t, t1, t2, t3, Char)
> -- equ: \a b c -> (a,1,b,c,'a')
```

Lambda Case

- Allows shorthand for inline case expressions

```
\case ->
    Red -> ...
    Green -> ...

-- previously
\color -> case color of
    Red -> ...
    Green -> ...
```

Multi-way If

- Better support for multi-branch `if` expressions using guard syntax
- Can be nested

```
if | HpPct > 75 -> Healthy
   | HpPct > 50 -> Standing
   | HpPct > 25 -> Faltering
   | HpPct > 0  -> Running
   | otherwise  -> Nope
```

Named Field Puns

- Short-hand for accessing record fields in a pattern match

```haskell
data D = D {a :: Int, b :: Int}

-- old way
f (D {a=a, b=b}) = a + b

-- new way
f (D {a, b}) = a + b
```

Record Wild Cards

- Brings all fields for a data type into scope by their accessor name

```
f (D{..}) = a + b
```

Explicit Namespaces

- Allows disambiguation between what namespace a `name` is supposed to come from
- Valid options: `pattern`, `family`, `type`
  - Everything else is found in the term namespace

```
import X.Y (pattern f, type (++), family Z, concatWith)
```

Bang Patterns

- Lightweight strictness annotations preceding
- Overrides default non-strictness
- Conventional wisdom: "strict leaves, lazy spine"
- Matching against strict `bottom` (`undefined` and kin) diverges/crashes

```
data Account = Account { !name :: String
                       , !aId :: Int
                       }
```

Typed Holes & Partial Type Signatures

- Interactive programming with Haskell!
- Typed Holes (7.8): in a function definition, insert _ to get an informative type **error**
- Partial Type Signatures (7.10): in a signature, insert _ to get a **warning** and an inferred type

```
id' a = _

Found hole ‘_’ with type: t1
Where: ‘t1’ is a rigid type variable bound by
            the inferred type of
    id' :: t -> t1 at <interactive>:2:5
Relevant bindings include
  a :: t (bound at <interactive>:2:9)
  id' :: t -> t1 (bound at <interactive>:2:5)
  In the expression: _
In an equation for ‘id'’: id' a = _
```

Typed Holes & Partial Type Signatures

```
{-# LANGUAGE PartialTypeSignatures #-}
id' :: _
id' a = a

H.hs:4:8: Warning:
  Found hole ‘_’ with type: t -> t
  Where: ‘t’ is a rigid type variable bound by
  the inferred type of
    id' :: t -> t at H.hs:5:1
In the type signature for ‘id'’: _
```

Derive Functor, Foldable, and Traversable

- Extend `deriving` to allow filling in:
  - **Functor**: `DeriveFunctor`
  - **Foldable**: `DeriveFoldable`
  - **Traversable**: `DeriveTraversable`
- There's a few others, but with more nuances than these

Overloaded Strings and Lists

- Allows for `String` literals to take on other types
- A generalized from exists to allow treating list literals as other types
- Actual type determined by surrounding context
- Compile-time **error** if ambiguous

```
class IsString a where
    fromString :: String -> a

instance ByteString a where
    fromString = pack
```

Overloaded Strings and Lists

```
class IsList a where
    type Item a
    fromList :: [Item l] -> l
    toList :: l -> [Item l]

instance (Ord a) => IsList (Set a) where
    type Item (Set a) = a
    fromList = Set.fromList
    toList = Set.toList
```

Bigger Extensions

- Let's try to understand some heavier extensions that change what can be expressed in Haskell.

Bigger Extensions

- Let's try to understand some heavier extensions that change what can be expressed in Haskell.
- GADTs

Bigger Extensions

- Let's try to understand some heavier extensions that change what can be expressed in Haskell.
- GADTs
- Existential quantification

Bigger Extensions

- Let's try to understand some heavier extensions that change what can be expressed in Haskell.
- GADTs
- Existential quantification
- Generalized Newtype Deriving

Bigger Extensions

- Let's try to understand some heavier extensions that change what can be expressed in Haskell.
- GADTs
- Existential quantification
- Generalized Newtype Deriving
- Empty Data Declarations

Bigger Extensions

- Let's try to understand some heavier extensions that change what can be expressed in Haskell.
- GADTs
- Existential quantification
- Generalized Newtype Deriving
- Empty Data Declarations
- Scoped Type Variables

GADTs

- A new syntax and semantics for modeling data types
- The key idea is that **pattern matching causes refinement**
- I'll walk through an example given in the user guide to illustrate this
- In practice, it means that given a polymorphic sum type:
  - Each branch can carry information about what **should** be there
  - GADTs let us model this

GADTs

```haskell
-- with GADTs
data Term a where
    Lit :: Int -> Term Int
    Succ :: Term Int -> Term Int
    IsZero :: Term Int -> Term Bool
    If :: Term Bool -> Term a -> Term a -> Term a
    Pair :: Term a -> Term b -> Term (a,b)
```

GADTs

```haskell
-- w/o GADTs
data Term
  = Lit Int
  | Succ Term
  | IsZero Term
  | If Term Term Term
  | Pair Term Term
```

GADTs

```
-- without GADTs, our language goes wrong
> :t If (Lit 10) (Lit 11) (Lit 12)
If (Lit 10) (Lit 11) (Lit 12) :: Term
```

GADTs

```
-- with GADTs, type-checker catches error
> :t If (Lit 10) (Lit 11) (Lit 12)
Couldn't match type ‘Int’ with ‘Bool’
   Expected type: Term' Bool
     Actual type: Term' Int
In the first argument of ‘If'’, namely ‘(Lit' 10)’
In the expression: If' (Lit' 10) (Lit' 11) (Lit' 12)
```

GADTs

```haskell
-- GADTs facilitate writing DSL interpreters
eval :: Term a -> a
eval (Lit i) = i
eval (Succ i) = 1 + eval i
eval (IsZero t) = eval t == 0
eval (If p l r) = if (eval p) then (eval l) else (eval r)
eval (Pair a b) = (eval a, eval b)
```

GADTs

- Some important limitations
  - Can no longer use `deriving`: requires `StandaloneDeriving`
  - May interfere with type inference
- When pattern matching, scrutinee, case expr result, and locals must be **rigid**
  - **rigid**: compiler knows type of term under consideration
  - may require type annotations from time to time

Existential Quantification

- Key intuition: homogeneous processing of heterogeneous collections

Existential Quantification

Existential Quantification

Existential Quantification

Existential Quantification

Generalized Newtype Deriving

- Mixes `newtype` with `deriving`
- Can `derive` any typeclass underlying type has instances for
- Allows for greater reuse of generative types

Generalized Newtype Deriving

Generalized Newtype Deriving

Generalized Newtype Deriving

Empty Data Declarations

- Allows for nullary data declarations
- Fantastic for phantom type techniques!

Empty Data Declarations

```
data OpenAccess
data Sensitive
data ReallySensitive
data TopSecret
```

Empty Data Declarations

Scoped Type Variables

- Allows for type annotations in more places than previously allowed
  - Scope of type signatures variables was previously limited to extent of type signature
- Further, these annotations represent **rigid** type **variables**
  - They're more than just hints
- Specifically, scoped type variables may be bound by:
  - A declaration type signature
  - An expression type signature
  - A pattern type signature
  - Class and instance declarations

Scoped Type Variables

```
f :: [a] -> [a]
f (x:xs) = xs ++ [ x :: a ] -- rejected by compiler
                            -- `a` not in scope

-- ExplicitForAll + ScopedTypeVariables
g :: forall a. [a] -> [a]
g (x:xs) = xs ++ [ x :: a ] -- okay!
```

Scoped Type Variables

- Frequently seen in conjunction with exception handlers:

```
try httpRequest uri `catch` (
  (\(e :: SomeException) -> print "Oh no...")
)
```

Dangerous Combinations

- GND and GADTs togther, in recent Haskell version

Next Steps

Resources

- Haskell 2010 Specification
- GHC/Haskell 7.10 User Guide
  - Scoped Type Variables
  - Empty Data Declarations
  - Existential Quantification
  - GADTs
  - Generalized Newtype Deriving

Thank You!