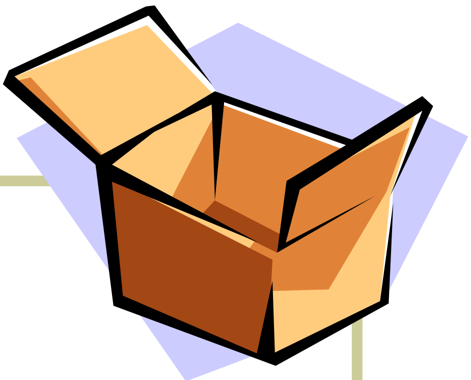
*Instances for Free**



Neil Mitchell

www.cs.york.ac.uk/~ndm

(* Postage and packaging charges may apply)

Haskell has type classes

```
f :: Eq a => a -> a -> Bool
f x y = x == y
```

- Polymorphic (for all type a)
- Provided they support Eq

Defining Type Classes

```
class Eq a where
    (==) :: a -> a -> Bool

data MyType = ...
instance Eq MyType where
    a == b = ...
```

Some Eq instances (1)

```
data List a = Nil | Cons a (List a)

instance Eq a => Eq (List a) where
    Nil == Nil = True
    Cons x1 x2 == Cons y1 y2 =
         x1 == y1 && x2 == y2
    _ == _ = False
```

Some Eq instances (2)

```
data Maybe a = Nothing | Just a

instance Eq a => Eq (Just a) where
    Nothing == Nothing = True
    Just x1  == Just y1 = x1 == y1
    _ == _ = False
```

Some Eq instances (3)

```
data Unit = Unit

instance Eq Unit where
    Unit == Unit = True
    _ == _ = False
```

Some Eq instances (4)

```
data Either a b = Left a | Right b

instance (Eq a, Eq b)
    => Eq (Either a b) where
    Left x1 == Left x2 = x1 == x2
    Right x1 == Right x2 = x1 == x2
    _ == _ = False
```

Please, no more Eq instances!

- In the base library there are 433 types with Eq instances
  - A lot of tedious code

- Fortunately, Haskell has a solution

```
data MyType = … deriving Eq
```

Limitations of “deriving”

- Can only derive 6 classes
  - Eq, Ord, Enum, Bounded, Show, Read
- But there are lots more classes out there

```
class Serial a where
    series :: Series a
    coseries :: Series b -> Series (a->b)
```

Solution: A preprocessor

- DrIFT was the original solution
  - Run a preprocessor to generate the instances
- Derive is a competitor to DrIFT
  - Can directly integrate with GHC
  - Preprocessor optional
  - Can work better with version control
  - Supports more Haskell features

How DrIFT works

- Has a representation of Haskell types
  - data HsType = HsType [Variable] [HsCtor]
  - data HsCtor = HsCtor CtorName [HsField]
- Author of the class must define
  - deriveSerial :: HsType -> String

How Derive works

- And a representation of Haskell code too
  - data Stmt = …
  - data Expr = …
  - Lots of constructors, types etc.

- Author of the class must define
  - `deriveSerial :: HsType -> [Stmt]`

Derive difficulties

- So the author of the Serial class must
  - Learn and understand HsType
  - Learn and understand Stmt, Expr etc.
  - Write an instance generator
  - Check it on several examples

- Lots of work for Colin!

A simpler solution

- Give Colin a single data type
  - Ask for a sample instance

```
data DataName a = CtorZero
                | CtorOne a
                | CtorTwo a a
                | CtorTwo' a a
```

Colin replies

```
instance Serial a =>
        Serial (DataName a) where
    series = cons0 CtorZero \/
             cons1 CtorOne \/
             cons2 CtorTwo \/
             cons2 CtorTwo'
```

```
data DataName a
  = CtorZero
  | CtorOne a
  | CtorTwo a a
  | CtorTwo' a a
```

Derive replies

```
[instance' [Serial] Serial
    [series = foldr1' (\/)
        [(cons +$ arity c) (name c)
        | c <- ctors]
    ]
]
a +$ b = a ++ show b
```

```
instance Serial a =>
        Serial (DataName a) where
    series = cons0 CtorZero \/
             cons1 CtorOne \/
             cons2 CtorTwo \/
             cons2 CtorTwo'
```

Derivation by Example

- We gave Derive one single example
  - Over a particular data type
- Derive has a domain specific language for instances
- Given an example, it infers a program

Instance for Eq

```
instance Eq a => Eq (DataName a) where
    CtorZero == CtorZero = True
    (CtorOne x1) == (CtorOne y1) =
        x1 == y1 && True
    (CtorTwo x1 x2) == (CtorTwo y1 y2) =
        x1 == y1 && x2 == y2 && True
    (CtorTwo’ x1 x2) == (CtorTwo’ y1 y2) =
        x1 == y1 && x2 == y2 && True
    _ == _ = False
```

Instance Example

```
[instance' [Eq] Eq
  [
    [(name c) [x +$ i | i <- [1..arity c]] ==
    [(name c) [y +$ i | i <- [1..arity c]] =
    foldr' (&&) True [x+$ i == y+$ i | i <- [1..arity c]]
  | c <- ctors]
  ++ [ _ == _ = False ]
]
```

What is in the Derive Language?

- map, foldr, foldl, foldr1, foldl1, reverse
- +$, ++
- ctors
- arity, name, tag
  - Properties over a constructor
- numbers
- instance'

Instance Derivation by Example

- Given an example for the data type
- Infer an instance

- **Key property**:
- If a derivation program is correct
- It must be equivalent to all other correct derivations

Uniqueness

- If only minimal derivations are considered, then the derivations are *unique*
  - Minimal = no redundant operations
- Achieved by bounding the domain language and selecting the data type

Example of Uniqueness

- For Serial, constructors map to *arity*
- For Enum, constructors map to *tags*

cons2 CtorTwo \/ cons2 CtorTwo’

fromEnum CtorTwo = 2

fromEnum CtorTwo’ = 3

Limitations

- Can’t deal with:
  - Records
  - Type based derivations
- Derive language cannot express these
- If they were added, the data type would have to become more complex (a lot!)

Summary so far…

- Derive lets you write one example
- Infers the pattern
- Works a lot of the time (~ 60%)

- Next
  - Basic idea behind the inference
  - Gets more technical…

Develop a “theory”

- The inference is bottom up
- Develops theories about syntactic bits
- Combines these theories

```
CtorZero  →  (\i -> name i)  CtorZero
CtorOne   →  (\i -> name i)  CtorOne
```

More theories

cons0	→	(\i -> cons +$ i)	0
cons1	→	(\i -> cons +$ i)	1
/\	→	(_ -> /\)	()

- Theories are parameterised by (), number, or a constructor

Promoting theories

theory () → theory <anything>

theory 0 → (theory . arity) CtorZero

theory 0 → (theory . tag) CtorZero

Nondeterministic

(\i -> cons +$ i) 0

→ (\i -> cons +$ arity i) CtorZero

→ (\i -> cons +$ tag i) CtorZero

Theory application

$$\frac{x \rightarrow x'\ t \qquad f \rightarrow f'\ t}{\begin{aligned} f\ x &\rightarrow (\backslash t \text{ -> } (f'\ t)\ (x'\ t))\ t \\ &\rightarrow (f' \text{ <*> } x')\ t \end{aligned}}$$

The S combinator

Theory lists

xi → xi' ti
forall i . xi' ti == xj' t'

In practice, all xi are usually identical

tn → expand t
forall i . (expand t !! i) == ti

[x1..xn] → (map xj' . expand) t

Theory expansions

n → (enumFromTo 1) n

→ (enumFromTo 0) n

CtorTwo' → ctors ()

[1,2,3] → (map id . enumFromTo 1) 3

[CtorZero..CtorTwo'] → (map id . ctors) ()

Adding in folds

- Just do a translation first

```
x1 `f` ... `f` xn == foldr1 f [x1...xn]
```

- Reverse is handled in the same way

Combined together

cons0 CtorZero

cons0 → (\i -> cons +$ i) 0

→ (\i -> cons +$ arity i) CtorZero

CtorZero → (\i -> name i) CtorZero

cons0 CtorZero →

(\i -> (name i) (cons +$ arity i)) CtorZero

More combining

```
[cons0 CtorZero, cons1 CtorOne, →
[(\i -> (name i) (cons +$ arity i)) CtorZero
,(\i -> (name i) (cons +$ arity i)) CtorOne
   →
(map (\i -> (name i) (cons +$ arity i))
  . ctors) ()
```

Conclusions

- The inference method is not *too* hard
- Usually just does the right thing

- If you really want to derive Serial, see:
  - http://www-users.cs.york.ac.uk/~ndm/derive/