Deriving Generic Functions by Example

Neil Mitchell

www.cs.york.ac.uk/~ndm/derive
“Generic” Functions

- Operates on *many* data types
- Think of equality
  - Comparing two integers, booleans, lists, trees
- Usually, must give each version separately
  - True in Ada, Java, Haskell...
- But usually they follow a pattern!
A Haskell data type

data List = Cons Int List
    | Nil

Cons 1 (Cons 2 (Cons 3 Nil))
Generic Equality on Lists

```haskell
instance Eq List where
    Cons a b ≡ Cons x y = a ≡ x ∧ b ≡ y
    Nil l ≡ Nil l = True
    _ _ ≡ _ _ = False
```

1. They have the same constructor
2. All fields are equal
Generic Equality on Trees

```haskell
data Tree = B Tree Int Tree | L

instance Eq Tree where
    B a b c ≡ B x y z = a ≡ x ∧ b ≡ y ∧ c ≡ z
    L       ≡ L       = True
    _       ≡ _       = False
```
Generic Equality on anything?

- Always follows the same simple pattern
- But highly dependent on the data type
  - If data type changes, updates required
  - Could “miss” a field doing it by hand
- Solution: Have it automatically generated
  - The DrIFT and Derive tools allows this
The Problem

- Need to state to the computer the relationship between data and code
  - Must be 100% precise
- I explained mainly through examples
- Requires learning an API, working at a meta-level, testing etc.
Specifying the Relationship

$\text{DataType} \rightarrow \text{String}$

eq' dat = simple_instance "Eq" dat [funN "==" body]
where
  body = map rule (dataCtors dat) ++
    [defclause 2 false]

rule ctor = sclause [ctp ctor 'a', ctp ctor 'b']
(and_ (zipWith (==) (ctv ctor 'a') (ctv ctor 'b')))

YUK!
Generic Functions by Example

- What if we provide only an example
  - The computer can infer the rules

- Uses concepts the user understands
- Guaranteed to work on at least 1 example
- Guaranteed to be type correct
- Quicker to write
Giving an example

- Needs to be on an *interesting* data type
  - Complex enough to have variety

```r
data DataName = First
  | Second  Any
  | Third   Any Any
  | Fourth  Any Any
```
And the example...

```haskell
instance Eq DataName where
  First ≡ First  = True
  Second x₁ ≡ Second y₁ = x₁ ≡ y₁ ∧ True
  Third x₁ x₂ ≡ Third y₁ y₂ = x₁ ≡ y₁ ∧ x₂ ≡ y₂ ∧ True
  Fourth x₁ x₂ ≡ Fourth y₁ y₂ = x₁ ≡ y₁ ∧ x₂ ≡ y₂ ∧ True
  _ ≡ _ = False
```

Now \(y_1\) and \(y_2\) instead of \(x\) and \(y\)

Redundant True at the end
Notation for Substitution

York $\Rightarrow$ Hello [#] $\Rightarrow$ Hello York
Tom $\Rightarrow$ Hello [#] $\Rightarrow$ Hello Tom
YDS $\Rightarrow$ [date] $\Rightarrow$ 2007/10/26
Assign Parameters

Idea: Move from the specific example, to a generalised version

Third

$y_1$, $y_2$

$3^{rd}$ constructor $\Rightarrow$ [name]

1 $\Rightarrow$ y[#]

2 $\Rightarrow$ y[#]
Group lists (MAP)

Only if:
1. Consecutive parameters
2. Same generator
The meaning of MAP

- \( 2 \Rightarrow MAP\ 1..\#\ y[\#] \)
- \( MAP\ 1..2\ y[\#] \)
- \( (1 \Rightarrow y[#]) \ (2 \Rightarrow y[#]) \)
- \( y_1\ y_2 \)
Generalise Numbers

$3^{rd}$ constructor

$2 \Rightarrow \text{MAP } 1..\# \ y[\#]$

$\text{Third}$

$y_1 \quad y_2$

$3^{rd}$ constructor

$\Rightarrow \text{[name]}$

$3^{rd}$ constructor

$\Rightarrow \text{MAP } 1..\text{arity } y[\#]$
Combine elements

3rd constructor ⇒ [name]

Third \( y_1 \) \( y_2 \)

3rd constructor ⇒ MAP 1..arity \( y[#] \)

3rd constructor ⇒ [name] (MAP 1..arity \( y[#] \))
Applying to other constructors

3\textsuperscript{rd} constructor
⇒ [name] (MAP 1..arity \ y[#])

First \rightarrow First
Second \rightarrow Second \ y_1
Third \rightarrow Third \ y_1 \ y_2
Fourth \rightarrow Fourth \ y_1 \ y_2
The Complete Generalisation

instance Eq [dataname] where

  [ MAP ctors (  
    ([name] [ MAP 1..arity (x[#])) ≡  
    ([name] [ MAP 1..arity (y[#])) =  
    [ FOLDR (∧) True  
      [ MAP 1..arity (x[#] ≡ y[#])]  
    ]  
  )]  

_ ≡ _ = False
Limitations: Non-inductive

- Example: Binary serialisation
  - Write out a tag (which constructor) then the fields
    - If only one constructor, no need for a tag
  - There is no general pattern
Limitations: Type Based

- Example: Monoid

- The instance for a Monoid is based on the types of the fields
  - Equal types have one value, different another
- The DataName type does not have different types
Limitations: Records

- Example: Show

```haskell
data Pair = Pair {fst :: Int, snd :: Int}
show (Pair 1 2) = "Pair {fst=1, snd=2}"
```

- Show includes the record field names
- DataName does not have record fields
Success Rate

- Eq
- Ord
- Data
- Serial
- Arbitrary
- Enum
- ...

[Pie chart showing 15 successful cases and 9 failure cases]
Future Work

- Extend the data type with more variety
  - Allows more classes to be specified
  - But more work to specify each class
- New uses for the information
  - Can derive classes at runtime
- Implement in other languages (Java?)
Conclusion

- Writing generic functions is cumbersome
- Writing generic relationships is hard
- Writing a single example is much easier
  - Works well in practice
  - Enables new contributors
Example 2

\[ 1 \Rightarrow x[\#] \quad 1 \Rightarrow y[\#] \]

\[ x_1 \equiv y_1 \land x_2 \equiv y_2 \land \text{True} \]

\[ 1 \Rightarrow x[\#] \equiv y[\#] \]

\[ 2 \Rightarrow x[\#] \equiv y[\#] \]

\[ \Rightarrow \text{True} \]
Generalising to a FOLDR

\[ 1 \implies x[#] \equiv y[#] \quad 2 \implies x[#] \equiv y[#] \quad \implies True \]

\[ x_1 \equiv y_1 \land x_2 \equiv y_2 \land True \]

\[ \text{FOLDR} (\land) True \]
\[ (1 \implies x[#] \equiv y[#], 2 \implies x[#] \equiv y[#]) \]
Generalising to a MAP

\[ FOLDR (\land) \text{ True} \]
\[ (1 \Rightarrow x[#] \equiv y[#], 2 \Rightarrow x[#] \equiv y[#]) \]

\[ x_1 \equiv y_1 \land x_2 \equiv y_2 \land \text{True} \]

\[ 2 \Rightarrow FOLDR (\land) \text{ True} \]
\[ (MAP \ 1..\# (x[#] \equiv y[#])) \]