Deriving Generic Functions by Example

Neil Mitchell

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

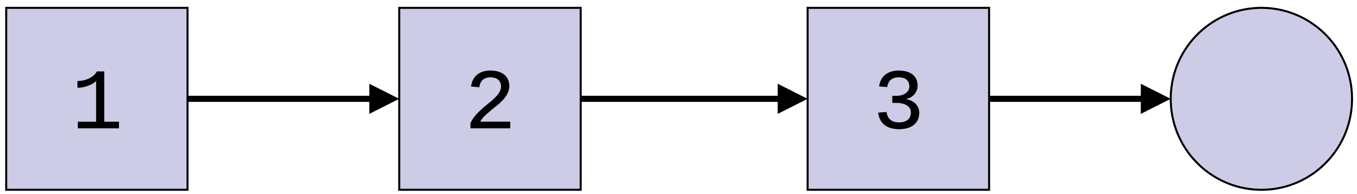
“*Generic*” Functions

- 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

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

1. They have the same constructor
2. All fields are equal

Generic Equality on Trees

```
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

```
DataType → 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

```
data DataName = First
              | Second Any
              | Third  Any Any
              | Fourth Any Any
```

And the example…

instance Eq DataName **where**

$\quad \texttt{First} \equiv \texttt{First} = \texttt{True}$

$\quad \texttt{Second}\ x_1 \equiv \texttt{Second}\ y_1 = x_1 \equiv y_1 \wedge \texttt{True}$

$\quad \texttt{Third}\ x_1\ x_2 \equiv \texttt{Third}\ y_1\ y_2 = x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge \texttt{True}$

$\quad \texttt{Fourth}\ x_1\ x_2 \equiv \texttt{Fourth}\ y_1\ y_2 = x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge \texttt{True}$

$\quad \_ \equiv \_ = \texttt{False}$

Now y_1 and y_2
instead of x and y

Redundant True
at the end

Notation for Substitution

York $\Rightarrow$ Hello [#] $\longrightarrow$ Hello York

Tom $\Rightarrow$ Hello [#] $\longrightarrow$ Hello Tom

YDS $\Rightarrow$ [date] $\longrightarrow$ 2007/10/26

Parameter

Substitute

Assign Parameters

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

`Third` y_1 y_2

3rd constructor $\Rightarrow$ [name]

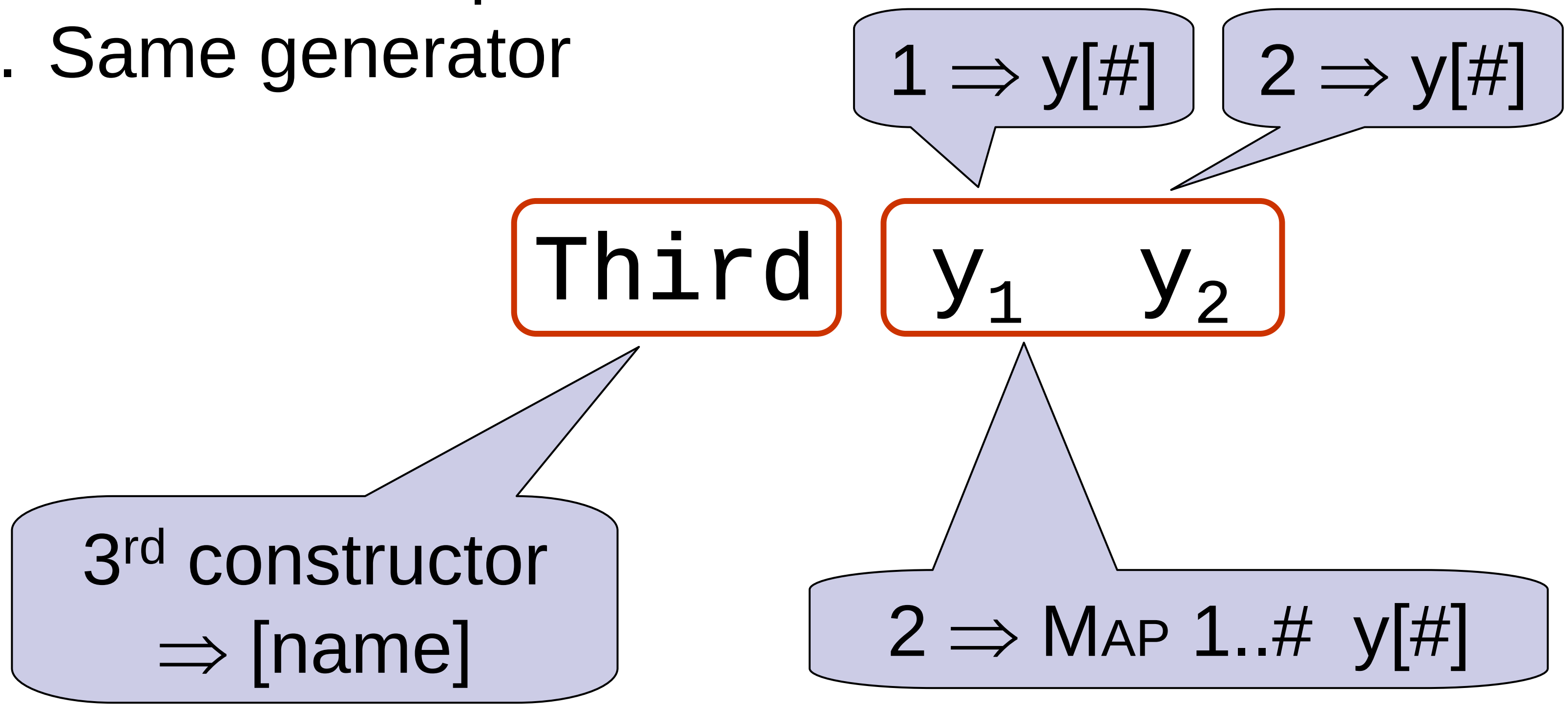
$1 \Rightarrow$ y[#]

$2 \Rightarrow$ 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

2 ⇒ MAP 1..# y[#]

`Third` y_1 y_2

3rd constructor ⇒ [name]

3rd constructor ⇒ MAP 1..arity y[#]

Combine elements

3rd constructor
$\Rightarrow$ [name]

3rd constructor
$\Rightarrow$ MAP 1..arity y[#]

`Third` y_1 y_2

3rd constructor
$\Rightarrow$ [name] (MAP 1..arity y[#])

Applying to other constructors

3rd constructor
$\Rightarrow$ [name] (MAP 1..arity y[#])

```
First   →  First
Second  →  Second y₁
Third   →  Third  y₁ y₂
Fourth  →  Fourth y₁ y₂
```

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

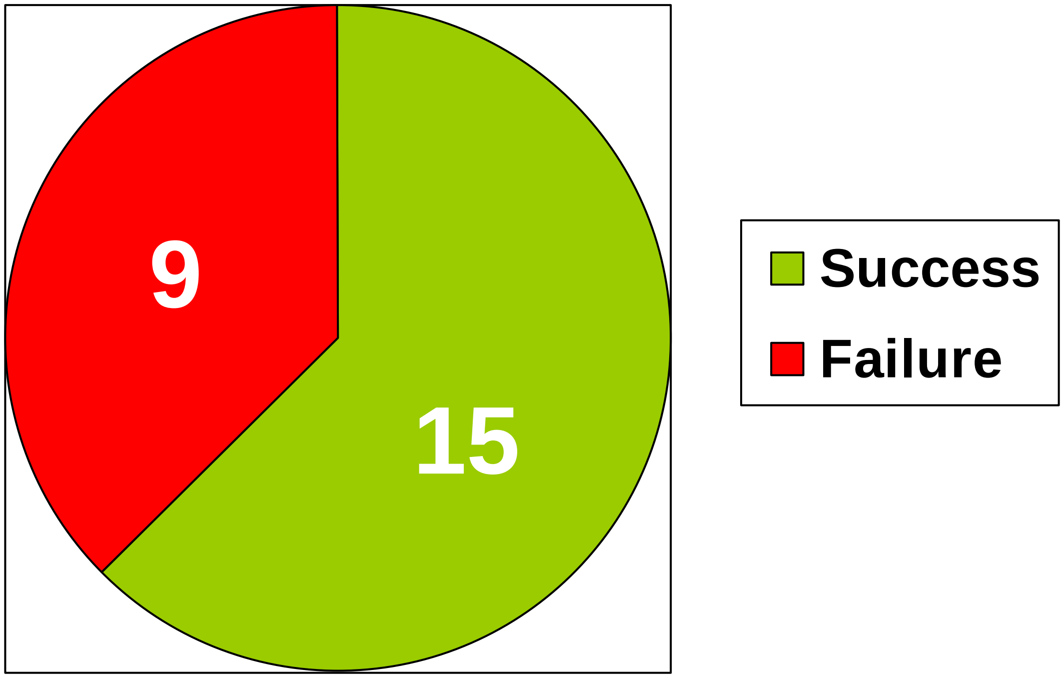
- Example: Show

```
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
- …



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

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

$$\boxed{x_1 \equiv y_1} \wedge \boxed{x_2 \equiv y_2} \wedge \boxed{\texttt{True}}$$

$1 \Rightarrow x[\#] \equiv y[\#]$ $\quad$ $\Rightarrow \text{True}$

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

Generalising to a Foldr

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

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

$\Rightarrow \text{True}$

$$\boxed{x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge \texttt{True}}$$

Foldr ($\wedge$) True

($1 \Rightarrow x[\#] \equiv y[\#]$, $2 \Rightarrow x[\#] \equiv y[\#]$)

Generalising to a Map

Foldr (∧) True
(1 ⇒ x[#] ≡ y[#], 2 ⇒ x[#] ≡ y[#])

$x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge \texttt{True}$

2 ⇒ Foldr (∧) True
(Map 1..# (x[#] ≡ y[#]))