Laws for Folds and Theorems for Free

Dave Laing

June 24, 2013

Outline

Outline

Outline

Outline

Preliminaries

Bottom

- We normally think of data types like this:

```
data Bool = False
          | True
```

Bottom

- We normally think of data types like this:

```
data Bool = False
          | True
```

- But in Haskell they're actually like this:

```
data Bool = False
          | True
          | ⊥
```

Strict and Non-Strict Functions

- A function f is strict if `f ⊥ = ⊥`
- Otherwise f is non-strict
- For example if

  ```
  f = const 2
  ```

  then

  ```
  f ⊥ = 2
  ```

Non-strict example

- Consider

```
and :: Bool -> Bool -> Bool
and False _   = False
and True  x   = x
and _     _   = ⊥
```

- Strict in the first argument
  - `and ⊥ x = ⊥`
- Non-strict in the second argument
  - `and False ⊥ = False`
  - `and True  ⊥ = ⊥`

Strict example

- Consider

```
and' :: Bool -> Bool -> Bool
and' False False = False
and' False True  = False
and' True  False = False
and' True  True  = True
```

- Strict in the first argument
  - and' ⊥ x = ⊥
- Strict in the second argument
  - and' x ⊥ = ⊥

Partial data

- $(\perp, \perp) \neq \perp$
  - A pair on the left, a... something... on the right
- $[\perp] \neq \perp$
  - It's a list with... something... in it
- $(1 : 2 : 3 : \perp)$ is a list with at least three elements in it
- Infinite lists are partial lists

Proofs with Partial Lists

- Prove $P([\,])$ and $P(xs) \rightarrow P(x : xs)$
  - P holds for all finite lists
- Prove $P(\perp)$ and $P(xs) \rightarrow P(x : xs)$
  - P holds for all partial lists
- Prove $P(\perp)$, $P([\,])$ and $P(xs) \rightarrow P(x : xs)$
  - P holds for all lists

Fold fusion

Fold Fusion

- When does
  `f . foldr g a = foldr h b` ?

Deriving the Fold Fusion Law

- Case $\perp$

Deriving the Fold Fusion Law

- Case $\perp$
- `f (foldr g a ⊥) = foldr h b ⊥`

Deriving the Fold Fusion Law

- Case $\perp$
- `f (foldr g a ⊥) = foldr h b ⊥`
- `f ⊥ = ⊥`

Deriving the Fold Fusion Law

- Case []

Deriving the Fold Fusion Law

- Case `[]`
- `f (foldr g a []) = foldr h b []`

Deriving the Fold Fusion Law

- Case `[]`
- `f (foldr g a []) = foldr h b []`
- `f a = b`

Deriving the Fold Fusion Law

- Case `(x:xs)`

Deriving the Fold Fusion Law

- Case `(x:xs)`
- Assume `f (foldr g a xs) = foldr h b xs`

Deriving the Fold Fusion Law

- Case `(x:xs)`
- Assume `f (foldr g a xs) = foldr h b xs`
- `f (foldr g a (x:xs)) = foldr h b (x:xs)`

Deriving the Fold Fusion Law

- Case `(x:xs)`
- Assume `f (foldr g a xs) = foldr h b xs`
- `f (foldr g a (x:xs)) = foldr h b (x:xs)`
- `f (g x (foldr g a xs)) = h x (foldr h b xs)`

Deriving the Fold Fusion Law

- Case `(x:xs)`
- Assume `f (foldr g a xs) = foldr h b xs`
- `f (foldr g a (x:xs)) = foldr h b (x:xs)`
- `f (g x (foldr g a xs)) = h x (foldr h b xs)`
- Let

```
y = foldr g a xs
```

Deriving the Fold Fusion Law

- Case `(x:xs)`
- Assume `f (foldr g a xs) = foldr h b xs`
- `f (foldr g a (x:xs)) = foldr h b (x:xs)`
- `f (g x (foldr g a xs)) = h x (foldr h b xs)`
- Let

  ```
  y = foldr g a xs
  ```
- Note that

  ```
  f(y) = f (foldr g a xs) = foldr h b xs
  ```

Deriving the Fold Fusion Law

- Case `(x:xs)`
- Assume `f (foldr g a xs) = foldr h b xs`
- `f (foldr g a (x:xs)) = foldr h b (x:xs)`
- `f (g x (foldr g a xs)) = h x (foldr h b xs)`
- Let

  `y = foldr g a xs`
- Note that

  `f(y) = f (foldr g a xs) = foldr h b xs`
- And so

  `f (g x y) = h x (f y)`

The Fold Fusion Law

So the conditions are

- `f` is strict
- `f a = b`
- `f (g x y) = h x (f y)`

Uses

- Reason about things that already exist
- Speed up your code
  - directly
  - via rules pragmas
  - let the compiler do it (but know it can happen)

Fold-Map Fusion

- Remember that

  ```
  map g = foldr ((:) . g) []
  ```

  so

  ```
  foldr f a . map g = foldr h b
  ```

  is subject to fold fusion.
- Fold fusion conditions imply that
  - `a = b`
  - `f . g = h`
- Resulting law is

  ```
  foldr f a . map g = foldr (f . g) a
  ```

Fold-Concat Fusion

- Remember that
  ```
  concat = foldr (++) []
  ```
  so
  ```
  foldr f a . concat
  ```
  is subject to fold fusion.
- Spolier alert:
  ```
  foldr f a . concat =  foldr (flip (foldr f)) a
  ```

Bookkeeping law

- Fold fusion works with map and with concat.
- So

  ```
  foldr f a . concat = foldr g b . map h
  ```

- Works out as

  ```
  foldr f a . concat = foldr f a . map (foldr f a)
  ```

- Examples:
  - `sum . concat = sum . map sum`
  - `concat . concat = concat . map concat`

Scans

- Like a fold that shows its working
- Uses `inits`
  - `inits [1,2,3] = [[],[1],[1,2],[1,2,3]]`
- `scanl f a = map (foldl f a) . inits`
- For example
  - `scanl (+) 0 [1..4] = [0, 1, 3, 6, 10]`
- Can work from the right as well
  - `scanr f a = map (foldr f a) . tails`

Fold-Scan Fusion

- Consider
  $1 + x_0 + x_0 * x_1 + x_0 * x_1 * x_2 + \ldots$
- Group terms
  $1 + x_0 * (1 + x_1 * (1 + x_2 * \ldots))$
- Haskellizes to
  `foldr1 (+) . scanl (*) 1`

Fold-Scan Fusion

- When does

  ```
  foldr1 (+) . scanl (*) e = foldr (⊙) e
  ```
- If
  - $*$ is associative with unit e
  - $*$ distributes over +
- Then

  ```
  foldr1 (+) . scanl (*) e = foldr (⊙) e
    where
      x ⊙ y = e + (x * y)
  ```

Maximum Subset Sum

- Problem posed and solved by Richard Bird
- Example of behaviour-preserving transformations from a trivially correct program to an efficient program
- Goal is to find the maximum sum of contiguous elements
- ```
  mss = maxlist . map sum . segs
    where
      maxlist = foldr1 max
      segs = concat . map inits . tails
  ```
  ```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
                tails [1, 2,-3] =
 [              [1, 2,-3]  ,
          [2,-3]  ,
      [-3]  ,
    []   ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
        map inits . tails $ [1, 2,-3] =
  [                    [1, 2,-3]   ,
               [2,-3]   ,
        [-3]   ,
     []   ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
      map inits . tails $ [1, 2,-3] =
[ [ [],[ 1],[1, 2],[1, 2,-3] ],
  [ [],[ 2],[2,-3] ],
  [ [],[-3] ],
  [ [] ] ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
concat . map inits . tails $ [1, 2,-3] =
  [ [ [],[ 1],[1, 2],[1, 2,-3] ],
    [ [],[ 2],[2,-3] ],
    [ [],[-3] ],
    [ [] ] ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
concat . map inits . tails $ [1, 2,-3] =
  [   [],[ 1],[1, 2],[1, 2,-3]   ,
      [],[ 2],[2,-3]   ,
      [],[-3]   ,
      []   ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
                    segs [1, 2,-3] =
  [   [],[ 1],[1, 2],[1, 2,-3]   ,
      [],[ 2],[2,-3]   ,
      [],[-3]   ,
      []   ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
           map sum . segs $ [1, 2,-3] =
[   [],[ 1],[1, 2],[1, 2,-3]   ,
    [],[ 2],[2,-3]   ,
    [],[-3]   ,
    []   ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
        map sum . segs $ [1, 2,-3] =
[    0,  1 ,    3 ,        0    ,
     0,  2 ,   -1    ,
     0, -3   ,
     0   ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
maxlist . map sum . segs $ [1, 2,-3] =
[     0,  1 ,     3 ,         0    ,
      0,  2 ,    -1    ,
      0, -3   ,
      0    ]
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
maxlist . map sum . segs $ [1, 2,-3] = 3
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

For example: `mss [1, 2, -3]`

```
mss [1, 2,-3] = 3
```

```
mss = maxlist . map sum . segs
  where
    maxlist = foldr1 max
    segs = concat . map inits . tails
```

Maximum Subset Sum

```
  mss
=    { definition of mss }
  maxlist . map sum . segs
```

Maximum Subset Sum

```
  maxlist . map sum . segs
=    { definition of segs }
  maxlist . map sum . concat . map inits . tails
```

Maximum Subset Sum

```
  maxlist . map sum . concat . map inits . tails
=    { bookkeepers law }
  maxlist . concat . map (map sum) . map inits . tails
```

Maximum Subset Sum

```
  maxlist . concat . map (map sum) . map inits . tails
=    { bookkeepers law }
  maxlist . map maxlist . map (map sum) . map inits . tails
```

Maximum Subset Sum

```
  maxlist . map maxlist . map (map sum) . map inits . tails
=    { functor law }
  maxlist . map (maxlist . map sum . inits) . tails
```

Maximum Subset Sum

```
  maxlist . map (maxlist . map sum . inits) . tails
=    { definition of scanl }
  maxlist . map (maxlist . scanl (+) 0) . tails
```

Maximum Subset Sum

```
  maxlist . map (maxlist . scanl (+) 0) . tails
=    { definition of maxlist }
  maxlist . map (foldr1 max . scanl (+) 0) . tails
```

Maximum Subset Sum

```
  maxlist . map (foldr1 max . scanl (+) 0) . tails
=    { fold-scan fusion }
     { + associative with unit 0 }
     { + distributes over max }
  maxlist . map (foldr addpos 0) . tails
    where
      addpos x y = 0 `max` (x + y)
```

Maximum Subset Sum

```
  maxlist . map (foldr addpos 0) . tails
=    { definition of scanr }
  maxlist . scanr addpos 0
```

Natural transformations

Definition

- Take two functors
  `F, G`
- Add a function
  `f :: A -> B`
- The natural transformation η makes
  $G(f) \circ \eta_A = \eta_B \circ F(f)$

What?

- Examples help
- F = `[[]]`
    - $F(f)$ = `map (map f)`
- G = `[]`
    - $G(f)$ = `map f`
- That means η = `concat` is a natural transformation between the list functor and the list-of-lists functor
    - `map f . concat = concat . map (map f)`

What?

- Examples help
- F = `[[]]`
  - $F(f)$ = `map (map f)`
- G = `[]`
  - $G(f)$ = `map f`
- That means η = `concat` is a natural transformation between the list functor and the list-of-lists functor
  - `map f . concat = concat . map (map f)`
- Bam!

What?

- Examples help
- F = `[[]]`
  - $F(f)$ = `map (map f)`
- G = `[]`
  - $G(f)$ = `map f`
- That means η = `concat` is a natural transformation between the list functor and the list-of-lists functor
  - `map f . concat = concat . map (map f)`
- Bam!
  - New law

More examples

- $F = G = []$
  - $F(f) = G(f) =$ `map f`
- Lots of options for η
  - `map f . reverse = reverse . map f`
  - `map f . tail = tail . map f`
  - `map f . init = init . map f`

More examples

- $F = G = []$
  - $F(f) = G(f) =$ `map f`
- Lots of options for η
  - `map f . reverse = reverse . map f`
  - `map f . tail = tail . map f`
  - `map f . init = init . map f`
- Bam! Bam! Bam!

More examples

- $F = G = []$
  - $F(f) = G(f) =$ `map f`
- Lots of options for η
  - `map f . reverse = reverse . map f`
  - `map f . tail = tail . map f`
  - `map f . init = init . map f`
- Bam! Bam! Bam!
  - New laws aplenty

More examples

- $F = G = []$
  - $F(f) = G(f) =$ `map f`
- Lots of options for η
  - `map f . reverse = reverse . map f`
  - `map f . tail = tail . map f`
  - `map f . init = init . map f`
- Bam! Bam! Bam!
  - New laws aplenty
- Any re-arrangement of the list will do

Parametericity

Polymorphic functions

- Consider
  `g :: forall A. [A] -> [A]`
- Could be `reverse`
- Could be `tail`
- Can only be a function that somehow rearranges the elements in the list

A free theorem appears

- We know that
```
map f . reverse = reverse . map f
```
- We also know that
```
map f . tail = tail . map f
```
- Actually true that
```
map f . g = g . map f
```
whenever
```
g :: forall A. [A] -> [A]
```

Polymorphism required

- Consider

```
g :: [Int] -> [Int]
g = filter odd
```

- and

```
f :: Int -> Int
f x = 2 * x
```

- Pretty clear that

  map f . g $\neq$ g . map f

- Stupid types, ruining stuff

The laws are *in* the types

- The laws can be derived mechanically
- Works for a variety of different typed functional languages
- Functions involved must be strict if `fix` is in the language

Relations on types

- The relationship between the sets A and $\acute{A}$ is denoted by $\mathcal{A} : A \Leftrightarrow \acute{A}$
- Equivalent to $\mathcal{A} \subseteq A \times \acute{A}$
- So $(x, \acute{x}) \in \mathcal{A}$ means x and $\acute{x}$ are related by $\mathcal{A}$
- If $t : T$ then $(t, t) \in \mathcal{T}$
- We can specialize the relation $\mathcal{A}$ to a function a : $A \rightarrow \acute{A}$
  - Then $(x, \acute{x}) \in a$ is the same as $a\ x = \acute{x}$

Building relations - pairs

- $((x, y), (\acute{x}, \acute{y})) \in \mathcal{A} \times \mathcal{B}$
  is equivalent to
  $(x, \acute{x}) \in \mathcal{A}$ and $(y, \acute{y}) \in \mathcal{B}$
- If we specialize $\mathcal{A}$ and $\mathcal{B}$ to functions a and b then
  $(a \times b)(x, y) = (a\ x, b\ y)$

Building relations - lists

- $([x_0, \ldots, x_n], [\acute{x}_0, \ldots, \acute{x}_n]) \in [\mathcal{A}]$
  is equivalent to
  $(x_0, \acute{x}_0) \in \mathcal{A}, \ldots, (x_n, \acute{x}_n) \in \mathcal{A}$
- If we specialize $\mathcal{A}$ to function a then
  $(xs, \acute{xs}) \in [a]$
  for all i, $a\ x_i = \acute{x}_i$
- Works out as
  $map\ a\ xs = \acute{xs}$

Building relations - from other relations

- $(f, \acute{f}) \in \mathcal{A} \to \mathcal{B}$
  is equivalent to
  for all $(x, \acute{x}) \in \mathcal{A}$, $(f\ x, \acute{f}\ \acute{x}) \in \mathcal{B}$
- If we specialize $\mathcal{A}$ and $\mathcal{B}$ to functions a and b then
  $(f, \acute{f}) \in a \to b$
  is equivalent to
  $\acute{f} \circ a = b \circ f$

Building relations - foralls

- $(g, \acute{g}) \in \forall \mathcal{X}.\mathcal{F}(\mathcal{X})$

  is equivalent to

  for all $\mathcal{A} : A \Leftrightarrow \acute{A}$, $(g_A, \acute{g}_{\acute{A}}) \in \mathcal{F}(\mathcal{A})$
- Means that for a relation through a forall, we add a relation and impose no restrictions on it
  - since it has to work for all relations we could have used

Rearranging functions

$g : \forall W.[W] \rightarrow [W]$

Rearranging functions

$(g, g) \in \forall \mathcal{W}.[\mathcal{W}] \to [\mathcal{W}]$

Rearranging functions

for all $\mathcal{X} : X \Leftrightarrow \acute{X}$

$(g_X, g_{\acute{X}}) \in [\mathcal{X}] \to [\mathcal{X}]$

Rearranging functions

for all $\mathcal{X} : X \Leftrightarrow \acute{X}$
  for all $(xs, \acute{xs}) \in [X]$
    $(g_A\ xs, g_{\acute{A}}\ \acute{xs}) \in [X]$

Rearranging functions

for all $f : X \to \acute{X}$

for all $xs, \acute{xs} \in [X]$

If $map\ f\ xs = \acute{xs}$

Then $map\ f\ (g_A\ xs) = g_{\acute{A}}\ \acute{xs}$

Rearranging functions

for all $f : X \to \acute{X}$

for all $xs \in [X]$

$$map\ f\ (g_A\ xs) = g_{\acute{A}}\ (map\ f\ xs)$$

$$map\ f \circ g_A = g_{\acute{A}} \circ map\ f$$

Fold law for free

- `foldr :: (X -> Y -> Y) -> Y -> [X] -> Y`
- If

  ```
  f (g x y) = h (e x) (f y)
  ```

  Then

  ```
  f . foldr g u = foldr h (f u) . map e
  ```

- Same as what we had before when e = id

Other cool stuff

- Can work with predicates
  - `sort :: (X -> X -> Bool) -> [X] -> [X]`
  - If

    ```
    (x < y) = (a x < a y)
    ```

    Then

    ```
    map a . sort (<) = sort (<) . map a
    ```
- Can work with polymorphic equality
  - So `Eq a => ...` typeclass constraints are allowed

Conclusion

- Laws can arise without typeclasses
- They can be handy
  - Handy to use
  - Handy to know
- Polymorphic types correspond to free theorems
- Free theorems correspond to natural transformations

Sources of things

- Theorems for Free by Philip Wadler
  - Builds on Reynolds' abstraction theorem
- Algebra of Programming by Bird and De Moor has more advanced stuff
- Slides at https://github.com/dalaing/bfpg-2013-06