Lawful Programming

How can we give useful information about a function without necessarily having to give all the details of its definition?

Informal description:
“map applies its first argument to every element in its second argument ...”

Type signature:
map :: (a -> b) -> [a] -> [b]

Laws:
- Normally in the form of equalities between expressions ...
Algebra of Lists

- $(++)$ is associative with unit $[]$
  
  $$xs ++ (ys ++ zs) = (xs ++ ys) ++ zs$$
  
  $$[] ++ xs = xs = xs ++ []$$

- map preserves identities, distributes over composition and concatenation:
  
  $$\text{map id} = \text{id}$$
  
  $$\text{map} \ (f \ . \ g) = \text{map} \ f \ . \ \text{map} \ g$$
  
  $$\text{map} \ f \ (xs ++ ys) = \text{map} \ f \ xs ++ \text{map} \ f \ ys$$
... continued

- **filter distributes over concatenation**
  - \( \text{filter } p (xs ++ ys) = \text{filter } p xs ++ \text{filter } p ys \)

- **Filters and maps:**
  - \( \text{filter } p . \text{map } f = \text{map } f . \text{filter } (p . f) \)

- **Composing filters:**
  - \( \text{filter } p . \text{filter } q = \text{filter } r \)
    where \( r x = q x \&\& p x \)
Aside: Extensionality

Two functions are equal if they give the same results on the same arguments

\[ f = g \iff \forall x. f(x) = g(x) \]

Example: \( f(x) = 1 + 2x \) and \( g = \text{((1+).(2*))} \), then:

\[
\begin{align*}
g(x) &= ((1+) \cdot (2*)) \cdot x \\
&= (1+) \cdot (2*) \cdot x \\
&= 1 + 2x \\
&= f(x)
\end{align*}
\]

Hence \( f = g \)
Laws Describe Interactions

A lot of laws describe how one operator interacts with another

Example: interactions with reverse:

- \( \text{reverse} \ . \ \text{map} \ f = \text{map} \ f \ . \ \text{reverse} \)
- \( \text{reverse} \ . \ \text{filter} \ p = \text{filter} \ p \ . \ \text{reverse} \)
- \( \text{map} \ f \ . \ \text{map} \ g = \text{map} \ (f \ . \ g) \)
- \( \text{reverse} \ (xs ++ ys) = \text{reverse} \ ys ++ \text{reverse} \ xs \)
- \( \text{reverse} \ . \ \text{reverse} = \text{reverse} \)

Caution: stating a law doesn’t make it true! (e.g., the last two laws for \text{reverse} are not true of all lists...)

Uses for Laws

Laws can be used:

- To capture/document deep intuitions about program behavior
- To support reasoning about program behavior
- To optimize or transform programs (either by hand, or in a compiler)
- As properties to be tested
- As properties to be proved
concat

concat :: [[a]] -> [a]
concat [[1,2], [3,4,5], [6]]  
= [1,2,3,4,5,6]

Laws:
- filter p . concat = concat . map (filter p)
- map f . concat = concat . map (map f)
- concat . concat = concat . map concat
Folds:
Folds!

A list $xs$ can be built by applying the $(::)$ and $[]$ operators to a sequence of values:

$$xs = x_1 :: x_2 :: x_3 :: x_4 :: ... :: x_k :: []$$

Suppose that we are able to replace every use of $(::)$ with a binary operator $(\oplus)$, and the final $[]$ with a value $n$:

$$xs = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus ... \oplus x_k \oplus n$$

The resulting value is called $\text{foldr} (\oplus) n xs$

Many useful functions on lists can be described in this way.
Graphically:

\[ f = \text{foldr} \ (\oplus) \ n \]
Example: sum

\[
\begin{align*}
\text{sum} &= \text{foldr} \ (+) \ 0
\end{align*}
\]
Example: product

\[
\text{product} = \text{foldr} \ (\ast) \ 1
\]
Example: length

\[
\text{length} = \text{foldr} \ (\lambda x \ ys \rightarrow 1 + ys) \ 0
\]
Example: map

\[
\begin{align*}
\text{map } f &= \text{foldr } (\lambda x \, ys \rightarrow f \, x : ys) \, [] \\
\end{align*}
\]
Example: filter

\[
\text{filter } p = \text{foldr } (\lambda x \ ys \to \begin{cases} p x & \text{then } x:ys \\ \text{else } ys & \end{cases}) \ [\]
\]
Formal Definition

foldr :: (a->b->b) -> b -> [a] -> b
foldr cons nil [] = nil
foldr cons nil (x:xs) = cons x (foldr cons nil xs)
Applications

\[
\begin{align*}
\text{sum} & \quad = \text{foldr } (+) \ 0 \\
\text{product} & \quad = \text{foldr } (*) \ 1 \\
\text{length} & \quad = \text{foldr } (\lambda x \ ys \rightarrow 1 + ys) \ 0 \\
\text{map } f & \quad = \text{foldr } (\lambda x \ ys \rightarrow f \ x : ys) \ [] \\
\text{filter } p & \quad = \text{foldr } c \ [] \\
\text{where } c \ x \ ys & \quad = \text{if } p \ x \ \text{then } x : ys \ \text{else } ys \\
\text{xs ++ ys} & \quad = \text{foldr } (:) \ ys \ xs \\
\text{and} & \quad = \text{foldr } (\&\&) \ True \\
\text{or} & \quad = \text{foldr } (||) \ False
\end{align*}
\]
Patterns of Computation

- `foldr` captures a common pattern of computations over lists
- As such, it is a very useful function in practice to include in the Prelude
- Even from a theoretical perspective, it is very useful because it makes a deep connection between functions that might otherwise seem very different ...
- From the perspective of lawful programming, one law about `foldr` can be used to reason about many other functions
A law about foldr

- If \((\oplus)\) is an associative operator with unit \(n\), then
  \[
  \text{foldr} (\oplus) n \text{ xs} \oplus \text{foldr} (\oplus) n \text{ ys} = \text{foldr} (\oplus) n (\text{xs ++ ys})
  \]

- \((x_1 \oplus ... \oplus x_k \oplus n) \oplus (y_1 \oplus ... \oplus y_j \oplus n) = (x_1 \oplus ... \oplus x_k \oplus y_1 \oplus ... \oplus y_j \oplus n)\)

- All of the following laws are special cases:
  - \(\text{sum xs} + \text{sum ys} = \text{sum (xs ++ ys)}\)
  - \(\text{product xs} \ast \text{product ys} = \text{product (xs ++ ys)}\)
  - \(\text{and xs} \&\& \text{and ys} = \text{and (xs ++ ys)}\)
  - \(\text{or xs} || \text{or ys} = \text{or (xs ++ ys)}\)
foldl

There is a companion function to foldr called foldl:

\[
\text{foldl} :: (b \to a \to b) \to b \to [a] \to b \\
\text{foldl} s n [] = n \\
\text{foldl} s n (x:xs) = \text{foldl} s (s n x) xs
\]

For example:

\[
\text{foldl} s n [e_1, e_2, e_3] \\
= s (s (s n e_1) e_2) e_3 \\
= ((n \cdot s \cdot e_1) \cdot s \cdot e_2) \cdot s \cdot e_3
\]
foldr vs foldl

foldr

foldl
Uses for foldl

Many of the functions defined using foldr can be defined using foldl:

\[
\begin{align*}
\text{sum} & = \text{foldl} \ (\ + \ ) \ 0 \\
\text{product} & = \text{foldl} \ (\ * \ ) \ 1
\end{align*}
\]

There are also some functions that are more easily defined using foldl:

\[
\begin{align*}
\text{reverse} & = \text{foldl} \ (\ \lambda \ ys \ x \ \rightarrow \ x:ys \ ) \ []
\end{align*}
\]

When should you use foldr and when should you use foldl? When should you use explicit recursion instead? ... (to be continued)
foldr1 and foldl1

- Variants of foldr and foldl that work on non-empty lists:
  - foldr1 :: (a -> a -> a) -> [a] -> a
  - foldr1 f [x] = x
  - foldr1 f (x:xs) = f x (foldr1 f xs)

- foldl1 :: (a -> a -> a) -> [a] -> a
  - foldl1 f (x:xs) = foldl f x xs

- Notice:
  - No case for empty list
  - No argument to replace empty list
  - Less general type (only one type variable)
Uses of foldl1, foldr1

From the prelude:

minimum = foldl1 min
maximum = foldl1 max

Not in the prelude:

commaSep = foldr1 (\s t -> s ++ ", " ++ t)
Example: Adding Commas

To make large numbers easier to read, it is common to insert a comma after every third digit, starting from the right.

Example: 1234567 -> “1,234,567”

The show function can turn an Integer into a String, but how do we insert the commas?
Example: Adding Commas

```
commas = reverse  
  . foldr1 (\s t -> s++","++t)  
  . group 3  
  . reverse  
  . show
```

```
"1,234,567"
"765,432,1"
["765", "432", "1"]
"7654321"
"1234567"
1234567
```
Summary

- Folds on lists have many uses

- Folds capture a common pattern of computation on list values

- In fact, there are similar notions of fold functions on many other algebraic datatypes ...
  - (Hence the Foldable type class...)

28