Map and Foldr

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science
Objectives

- Explain the concept of *first class citizen*.
- Show several ways to define functions in Haskell.
- Define the `foldr` and `map` functions.
- Use `foldr` and `map` to implement two common recursion patterns.
First Class Functions

An entity is said to be first class when it can be

- assigned to a variable, passed as a parameter, or returned as a result.

Examples:

- APL: scalars, vectors, arrays
- C: scalars, pointers, structures
- C++: like C, but with objects
- Haskell, Lisp, OCaml: scalars, lists, tuples, functions

The Kind of Data a Program Manipulates Changes the Expressive Ability of a Program
Defining Functions the Usual Way

Some Haskell Functions

```haskell
1  sqr a = a * a
2  hypotsq a b = sqr a + sqr b
```

Sample Run

```haskell
1  sqr :: Integer -> Integer
2  sqr :: Num a => a -> a
3  hypotsq :: Num a => a -> a -> a
4  Prelude> sqr 10
5    100
6  Prelude> hypotsq 3 4
7    25
```
Example: Compose

Example

```
inc x = x + 1
double x = x * 2
compose f g x = f (g x)
```

- Notice the function types.

```
compose :: (t1 -> t2) -> (t -> t1) -> t -> t2
Prelude> :t double
double :: Integer -> Integer
Prelude> double 10
20
Prelude> compose inc double 10
21
```
Example: Twice

One handy function allows us to do something twice.

Twice

```haskell
1 twice f x = f (f x)
```

Here is a sample run...

```
Prelude> :t twice
  twice :: (t -> t) -> t -> t
Prelude> twice inc 5
  7
Prelude> twice twice inc 4
```
Creating Functions: Lambda Form

- Functions do not have to have names.

\[
\lambda x \rightarrow x + 1
\]

- The Parts:
  - Backslash (a.k.a. \textit{lambda})
  - Parameter list
  - Arrow
  - Body of function

\[
\text{prelude}\triangleright (\lambda x \rightarrow x + 1)\ 41
\]

\[
42
\]
Creating Functions: Partial Application

Standard form vs. Anonymous form

```
inc :: (Num t) => t -> t
inc a = a + 1
inc = \a -> a + 1

plus :: (Num t) => t -> t -> t
plus a b = a + b
plus = \a -> \b -> a + b
```

▶ What do you think we would get if we called `plus 1`?
Creating Functions: Partial Application

Standard form vs. Anonymous form

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>inc :: (Num t) =&gt; t -&gt; t</td>
</tr>
<tr>
<td>2</td>
<td>inc a = a + 1</td>
</tr>
<tr>
<td>3</td>
<td>inc = \a -&gt; a + 1</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>plus :: (Num t) =&gt; t -&gt; t -&gt; t</td>
</tr>
<tr>
<td>6</td>
<td>plus a b = a + b</td>
</tr>
<tr>
<td>7</td>
<td>plus = \a -&gt; \b -&gt; a + b</td>
</tr>
</tbody>
</table>

▶ What do you think we would get if we called plus 1?

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>inc = plus 1</td>
</tr>
</tbody>
</table>
\( \eta \)-equivalence

An Equivalence

\[ f \equiv \lambda x \rightarrow f \ x \]

Proof, assuming \( f \) is a function...

\[ f \ z \equiv (\lambda x \rightarrow f \ x) \ z \]

These are equivalent

1. `plus a b = (+) a b`
2. `plus a = (+) a`
3. `plus = (+)`

So are these

1. `inc x = x + 1`
2. `inc = (+) 1`
3. `inc = (+1)`
Let’s talk about mapping.

Incrementing Elements of a List

```
1 incL [] = []
2 incL (x:xs) = x+1 : incL xs
```

incL [7,5,6,4,2,-1,8] ⇒ [8,6,7,5,3,0,9]
Mapping functions the hard way

What do the following definitions have in common?

Example 1

1. \( \text{incL} \; [] = [] \)
2. \( \text{incL} \; (x:\text{xs}) = x+1 \; : \; \text{incL} \; \text{xs} \)

Example 2

1. \( \text{doubleL} \; [] = [] \)
2. \( \text{doubleL} \; (x:\text{xs}) = x*2 \; : \; \text{doubleL} \; \text{xs} \)
Mapping functions the hard way

Example 1

1. \( \text{incL} \ [\ ] = [\ ] \leftarrow \text{Base Case} \)
2. \( \text{incL} \ (x:xs) = x+1 : \text{incL} \ xs \)

Recursion

Example 2

1. \( \text{doubleL} \ [\ ] = [\ ] \leftarrow \text{Base Case} \)
2. \( \text{doubleL} \ (x:xs) = x*2 : \text{doubleL} \ xs \)

Recursion

- Only two things are different:
  - The operations we perform
  - The names of the functions
Mattox’s Law of Computing

The computer exists to work for us; not us for the computer. If you are doing something repetitive for the computer, you are doing something wrong.

Stop what you’re doing and find out how to do it right.
Mapping functions the easy way

Map Definition

\[
map f [x_0, x_1, \ldots, x_n] = [f x_0, f x_1, \ldots, f x_n]
\]

```
1  map :: (a->b) -> [a] -> [b]
2  map f [] = []
3  map f (x:xs) = f x : map f xs

incL = map inc

doubleL = map double
```

- inc and double have been transformed into recursive functions.
- I dare you to try this in Java.
Let’s talk about folding

What do the following definitions have in common?

Example 1

1. \( \text{sumL} \left[ \right] = 0 \)
2. \( \text{sumL} \left( \text{x:xs} \right) = x + \text{sumL} \text{ xs} \)

Example 2

1. \( \text{prodL} \left[ \right] = 1 \)
2. \( \text{prodL} \left( \text{x:xs} \right) = x \times \text{prodL} \text{ xs} \)
**foldr**

**Fold Right Definition**

\[ \text{foldr } f \; z \; [x_0, x_1, \ldots, x_n] = f \; x_0 \; (f \; x_1 \; \cdots \; (f \; x_n \; z) \; \cdots) \]

- To use **foldr**, we specify the function *and* the base case.

1. \( \text{foldr} :: (a \to b \to b) \to b \to [a] \to [b] \)
2. \( \text{foldr } f \; z \; [] = z \)
3. \( \text{foldr } f \; z \; (x:xs) = f \; x \; (\text{foldr } f \; z \; xs) \)

4. \( \text{sumlist} = \text{foldr } (+) \; 0 \)
5. \( \text{prodlist} = \text{foldr } (*) \; 1 \)
Encoding Recursion using `fold`

- Notice the pattern between the recursive version and the higher order function version.

**Recursive Style**

1. `plus a b = a + b`
2. `sum [] = 0`
3. `sum (x:xs) = plus x (sum xs)`

```
sum = foldr (\a b -> plus a b) 0
```
Some things to think about…

▶ These functions scale to clusters of computers.
▶ You can write `map` using `foldr`. Try it!
▶ You cannot write `foldr` using `map` — why not?